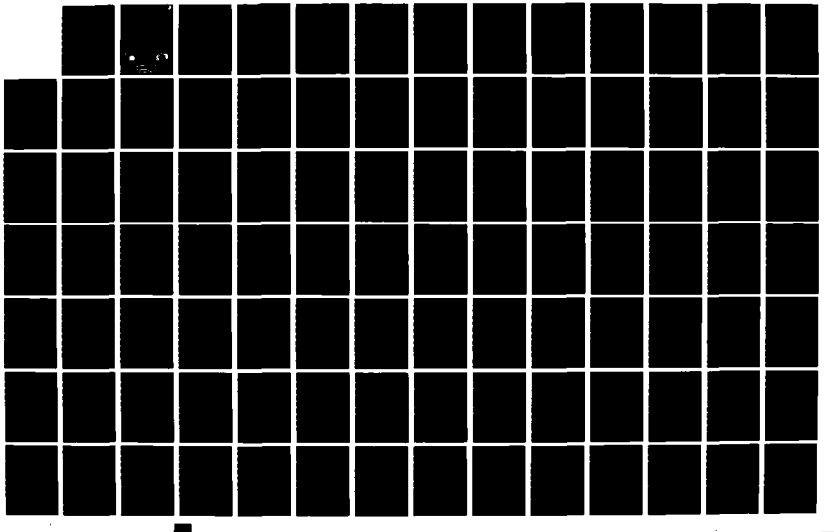
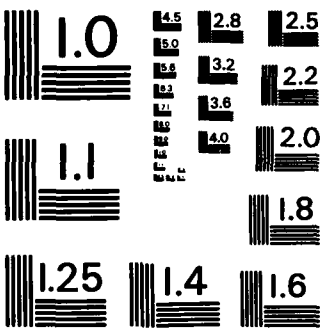
AD-A158 477 RESEARCH IN DISTRIBUTED TACTICAL DECISION MAKING: DECENTRALIZED RESOURCE..(U) CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE

UNCLASSIFIED E D JENSEN ET AL. 31 JUL 85 F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Research in Distributed Tactical Decision Making: Decentralized Resource Management in Tactical Computer Executives

SFRC: N00014-84-K-0734
Work Unit No. 649-004
Annual Report
1 August 1984 - 31 July 1985

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

Research in Distributed Tactical Decision Making: Decentralized Resource Management in Tactical Computer Executives

SFRC: N00014-84-K-0734
Work Unit No. 649-004
Annual Report
1 August 1984 - 31 July 1985

Prepared for

Department of the Navy

Office of Naval Research

800 North Quincy St., Arlington, VA 22217

by

Computer Science Department

Department of Electrical and Computer Engineering

Department of Statistics

Carnegie-Mellon University

Pittsburgh, Pennsylvania 15213

(412) 578-2574

This research was supported by the Engineering Psychology Program, Office of Naval Research. Approved for public release; distribution unlimited. Reproduction in whole or in part is permitted for any purpose of the United States Government.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS: NONE
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: Dept. of Computer Science; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: ONR
6c. ADDRESS (City, State and ZIP Code): Carnegie-Mellon University, Pittsburgh, PA 15213	7b. ADDRESS (City, State and ZIP Code): Pasadena, CA 91106
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Office of Naval Research; 8b. OFFICE SYMBOL (If applicable): 442EP	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: N00014-84-K-0734
8c. ADDRESS (City, State and ZIP Code): Arlington, VA 22217	10. SOURCE OF FUNDING NOS. — PROGRAM ELEMENT NO.; PROJECT NO.; TASK NO.; WORK UNIT NO.
11. TITLE (Include Security Classification): Decentralized Resource Management in Tactical Computer Executives (u)	
12. PERSONAL AUTHOR(S): E.D. Jensen, L. Sha, J.P. Lehoczky, C.D. Locke, S.E. Shipman, A.M. vanTilborg	
13a. TYPE OF REPORT: Annual Summary; 13b. TIME COVERED: FROM 84Aug01 TO 85Jul31; 14. DATE OF REPORT (Yr., Mo., Day): 85, Jul, 31; 15. PAGE COUNT: 98	
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	Best effort decision making, dynamic reconfiguration, decentralized scheduling, non-serializable transactions

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report summarizes first year progress on the subject contract. It presents an analysis of dynamic reconfiguration of distributed computing systems, and provides a set of powerful reconfiguration algorithms. The report also describes basic research into team decision making as applied to load balancing. In addition, the report contains verbatim copies of two technical papers on the subject of modular concurrency control and failure recovery.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Dr. David Mizell	22b. TELEPHONE NUMBER (Include Area Code): (818) 795-5971; 22c. OFFICE SYMBOL: ONR Code 433

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

Table of Contents

List of Figures

List of Tables

Report Distribution

Addressee	Copies
1. Office of Naval Research Head, Mathematical Sciences Division Code N00014 800 North Quincy St. Arlington, VA 22217	1
2. Office of Naval Research Resident Representative, Code N66002 National Academy of Sciences Joseph Henry Bldg., Rm. 623 2100 Pennsylvania Ave. NW. Washington, DC 20037	1
3. Office of Naval Research Code 433 Attn: Dr. David Mizell 1030 E. Green St. Pasadena, CA 91105	1
4. Space and Naval Warfare Systems Command Code 611 Attn: Dr. Kepi Wu Washington, DC 20363-5100	1

5. Office of Naval Research — 1
   Code 442EP
   Attn: Dr. William Vaughn, Jr.
   800 North Quincy St.
   Arlington, VA 22217

6. Director, Naval Research Laboratory — 1
   Attn: Code 2627
   Washington, DC 20375

7. Defense Technical Information Center — 12
   Bldg. 5
   Cameron Station
   Alexandria, VA 22314

Research in Distributed Tactical Decision Making:

Decentralized Resource Management

in

Tactical Computer Executives

This report prepared by:

E. Douglas Jensen, Principal Investigator

Lui Sha, John P. Lehoczky, C. Douglass Locke, Samuel E. Shipman, Investigators

Andre M. van Tilborg, Program Manager

1. Summary

1.1 Overview

This report summarizes first year technical progress on SFRC N00014-84-K-0734, "Research in Distributed Tactical Decision Making: Decentralized Resource Management in Tactical Computer Executives," for the period 1 August 1984 (contract start date) through 31 July 1985. This research reported herein is jointly funded by Office of Naval Research and Space and Naval Warfare Systems Command.

The objective of this effort is to conduct research in the area of distributed tactical decision making. Carnegie-Mellon University (CMU) proposed to conduct research on primarily two tasks in the first contract year. Task A considers best effort decision making for decentralized resource management. Task B focuses on the development of atomic transaction theories to support highly concurrent and reliable real-time computation. Innovative research under the guidance of the Principal Investigator was conducted on both of these tasks. Work will continue on both tasks in the second contract year.

1.2 Progress on Task A: Best Effort Decision Making

Task A is divided into three subtasks. Subtask A1 deals with the development of best effort decision making algorithms for dynamic system reconfiguration for the purpose of enhancing system survivability. Subtask A2 is a basic research task which investigates the theory of best effort team decision making. Subtask A3 investigates real-time multi-processor scheduling. In the following section, we summarize the progress made in Task A. A detailed discussion of these three subtasks is presented in Chapters 2 through 4.

1.2.1 Subtask A1: System Reconfiguration Decision Algorithms

CMU has developed a set of reconfiguration algorithms with the following properties:

- Disturbance to functioning processes during the process of reconfiguration is minimized,
- System vulnerability to future failures after re-configuration is minimized,
- Reconfiguration is performed in real-time.

These algorithms are directly applicable to a variety of distributed tactical systems.

1.2.2 Subtask A2: Best Effort Decision Theory

CMU has developed results showing that the lower bound on the performance of best effort team decision making in the context of load balancing is 62% of the idealized performance with full and instantaneous information. In this analysis, only local information is used for decision making. Since this result is the *lower* bound of best effort decision making in the context of load balancing, it demonstrates the promise of decentralized best effort team decision making. A protocol for the efficient use of active information has been investigated. Further research on this topic will be continued in the next year.

1.2.3 Subtask A3: Multi-processor Real-time Scheduling

Subtask A3 represents our initial work on a value function based approach for multi-processor real-time scheduling. This research involves the study of deadline management in a multi-processor environment, in which the time allocation decisions must be made using a best effort approach. A considerable amount of research has been done when deadlines could be met, but relatively little information is available about scheduling decisions when available resource limitations require that one or more deadlines cannot be met. Our approach is designed to maximize the value of the available state information to make the deadline scheduling decisions, particularly in those cases where deadlines cannot be met.

1.3 Progress on Task B: Atomic Transaction Theory

Task B is divided into two subtasks. Subtask B1 represents the conclusion of a long term research of atomic transaction theory for the development of transaction facilities in decentralized operating systems. Atomic transaction facilities are important for the reliable and efficient management of distributed data. Therefore, these facilities are also important for implementing any distributed decision making protocol. Subtask B2 initiates the investigation of co-operating transactions, transactions that share information and co-operate as a team to achieve some common goal. In the following section, we summarize the progress made in Task B. The detailed report of Task B is presented in Chapters 5 and 6.

1.3.1 Subtask B1: Atomic Transaction Theory

CMU's major achievement in the area of atomic transactions is the development of a formal theory of modular concurrency control and failure recovery. This theory is a generalization of classical serializability theory and failure atomicity. This theory provides the basis for both optimal modular concurrency control and optimal failure recovery rules. As long as each programmer ensures the consistency and correctness of his

ii. Label the identified node as the candidate node.

iii. List the candidate node in the merging list.

iv. In the graph, merge the candidate node with the merged node.

f. Take the first processor in the processor list. Repeat the following until the merging list is emptied.

i. Take the first node in the merging list, delete it from the list and try to fit it in the processor.

ii. Check all the constraints, real or pseudo.

iii. If successful, then this node is a resident of the processor.

iv. If a node becomes a resident of a processor, delete it from the communication graph.

g. Delete the processor from the processor list.

h. If the processor list is emptied but the communication graph is not, then report that the reconfiguration fails.

i. If the processor list is not emptied but the communication graph is, then delete the mode from the movable mode list.

2.6 The Reconfiguration Procedure

In this section, we first list the reconfiguration routine. Next, we list the overall reconfiguration procedure.

2.6.1 The Reconfiguration Routine

We list the reconfiguration routine.

1. Run the reverse FFD algorithm to obtain the movable mode list.

2. If it is a processor failure, then run the new multi-dimensional FFD algorithm. If this FFD algorithm fails, then report that the reconfiguration fails. In addition, report the modes that are still disabled and exit this routine.

3. Check the communication constraints.

4. If the communication constraints are satisfied, report to the operator that the reconfiguration is successful, and exit this routine.

a. List all the processes in the mode in decreasing order according to their maximal real resource requirement such as CPU, memory, ports etc. For example if a process needs 20% CPU and 10% memory, its maximal resource requirement is 0.2.

b. Take the first process in the process list and repeat the following until either the list is emptied or reconfiguration fails.

   i. Try to put the process in the first processor in the ordered processor list.

   ii. Check all the constraints in all the dimensions, real or pseudo.

   iii. If all the constraints are satisfied, then this process becomes the resident of the processor. In this case, delete the process from the list.

   iv. If the process cannot be fit into any processor then report to operator that reconfiguration fails.

c. If the list of processes is emptied, then delete the associated mode from the list of movable modes.

2.5.3 The Clustering Algorithm

The steps of our clustering algorithm are as follows:

1. List all the movable modes in decreasing order according to their linear criticality score.

2. List all the processors in decreasing order according their minimal available capacities as defined in the previous section.

3. Take the first mode in the list and repeat the following until either the list is emptied or the reconfiguration fails.

   a. Draw the communication graph of all the processes in the mode.

   b. Identify the maximal arc.

   c. Put one of the two nodes joined by the arc on a merging list.

   d. In the graph, mark the node on the merging list as the merged node.

   e. Repeat the following until all the nodes in the graph are put into the list.

      i. In the graph, identify the node which communicates most heavily with the merged node.

memory utilization 0.3 and 0.5, while these average are 0.4 and 0.3 for mode B, then mode A and mode B are said to be compatible. This is because the total utilization are 0.7 and 0.8, which are both less than 1. If a functioning mode is compatible with any of the failed modes, then it is said to be a candidate movable mode. The idea of compatibility is that if a failed mode is CPU intensive, then we want to match it with a mode which uses little CPU, so that they can be packed together later. We identify all the candidate movable modes.

5. For each candidate movable mode M_i, we compute its total resource supply R_i by summing up all the resources it utilizes in percentage of total units.

6. We now try to fill the pseudo bin by first putting all the failed modes into it. We then repeat the following steps until the list of candidate modes is emptied.

   a. For each of the candidate movable modes M_i, we compute the additional disturbance score D_i, which would arise if mode M_i is disturbed.

   b. For each of the candidate movable modes M_i, we compute the merit score R_i/D_i. The merit score is a measure of the amount of resource provided by a mode per unit disturbance incurred.

   c. We try to put the one with maximal merit score into the pseudo bin. Call this mode as the candidate bin resident and delete it from the candidate movable mode list.

   d. Add up the total disturbance scores of the bin resident modes and this candidate bin resident. If this total score is less than S_t, then the candidate becomes a resident of the pseudo bin.

7. Report all the modes in the pseudo bin as movable modes to the operator for approval and possible alteration.

2.5.2 The Modified Multi-dimensional FFD algorithm

The steps to carry out the multi-dimensional FFD algorithm are as follows:

1. Augment each processor with a set of pseudo joint criticality resources, each of which corresponds to a joint criticality index.

2. List all the movable modes in decreasing order according to their linear criticality score.

3. List all the processors in decreasing order according to their minimal capacities. For example, if processor A has 10% available CPU cycles and 20% available memory, then the minimal capacity is 10%.

4. Take the first mode in the list and repeat the following until either the list is emptied or the reconfiguration fails.

FFD algorithm. To avoid putting processes belonging to a set of jointly critical modes into a single processor, we require the clustering algorithm to observe the pseudo resource constraints associated with each processor. To promote the clustering of processes belonging to a same mode, we run the clustering algorithm on a mode by mode basis. That is, we examine the communication graph of each mode and try to cluster those processes with heavy inter-process communication first. Since processes belonging to the same mode co-operatively carry out a task, they tend to communicate with each other more than with processes of other modes. Thus the mode by mode clustering is likely to produce results nearly as good as the results of global clustering.

2.5 The Algorithmic System

In this section, we first outline in detail the reverse FFD algorithm, the modified FFD and clustering algorithms. Next, we present our overall procedure for carrying out the reconfiguration.

2.5.1 Maximizing The Feasibility: The Reverse FFD Algorithm

The Reverse FFD algorithm is used to maximize the reconfiguration feasibility by identifying all the movable modes with respect to the given allowable disturbance level S_a. That is, the algorithm takes the current configuration and the set of failed modes and produces a set of functioning modes which can be disturbed within the disturbance constraint S_a.

We now list the steps of this algorithm.

1. If S_a is zero, then list all the modes that have been disabled by the failure as movable modes and exit this procedure.

2. If S_a is non-zero, then create a pseudo bin whose size is $S_t = S_c + S_a$, where S_c is the current disturbance due to the failure.

3. For each mode, we compute the average process resource utilization for each resource type. For example, mode A has 3 processes which use 0.1, 0.2, and 0.3 units of CPU and 0.4, 0.5, 0.6 units of memory respectively, then in this case the average CPU utilization is $(0.1 + 0.2 + 0.3)/3 = 0.2$ units and the average memory utilization is 0.5 units.

4. For each of the failed modes, we pair it with each of the functioning modes and check their *joint compatibility*. If the sum of the average utilization of each of the resource types is less than one,[2] then the two modes are said to be compatible. For example, if mode A has average CPU and

[2] One is a generally good value for the threshold in this case. However, the value for the threshold can be experimentally tuned.

chosen to offer a high probability of finding a feasible reconfiguration. This algorithm is described in Section 2.3.1.

2.4.3 Controlling Vulnerability

We have developed our approach to keep the reconfiguration disturbance below a permitted level S_a. Given that this constraint is satisfied, we now must minimize the future vulnerability. As discussed before, we cannot hope to find the minimal future vulnerability configuration in real-time. However, we can take advantage of the characteristics of a given system and develop an approach that produces nearly optimal results.

When a processor fails, all the modes relying on the processes running on this processor are interrupted. Thus, to minimize future vulnerability, a key principle is to avoid putting multiple modes on a single processor, especially jointly critical modes. Another important aspect in minimizing future vulnerability is to try pack all the processes of a mode into as few processors as possible, because when a mode is supported by several processors, the failure of any of these processors interrupts the mode. Both of these aspects can be accomplished by modifying the multi-dimensional FFD algorithm in the linear phase of the reconfiguration procedure and the clustering algorithm in the quadratic phase.

In the linear phase, to avoid putting multiple critical modes into a single processor, we associated pseudo resources with each processor. Each of the pseudo resources corresponds to a joint critical index as discussed before. In the example we discussed earlier, the two scanning modes have a joint criticality score of 0.8. To prevent the processes of both scanning modes being put into a single processor, we associate one unit of pseudo scanning resource with each of the processors that can support scanning modes. In addition, we specified that a scanning mode requires 0.6 unit of the pseudo scanning resource, while other non-scanning modes requires none of this pseudo scanning resource. Thus, a processor can only support the processes of one of the two scanning modes, but not both. To put the processes of a mode into as few processors as possible, we first run the multi-dimensional FFD algorithm for the most significant mode with respect to the processors with most available capacities. This gives the best chance for the FFD algorithm to pack the most significant mode into the least number of processors. After packing the most significant mode, we repeat this processes for the second most important mode and so on.

In the quadratic phase, the modification of the clustering algorithm is similar to the modification of the

2.4.2 Controlling Disturbance

We have mentioned that we are in favor of keeping the reconfiguration disturbance S_a to a low level while trying to produce a configuration that is low in future vulnerability. We assume that the default value of S_a is zero. That is, without the operator's explicit permission, the only processes that are allowed to be relocated by the reconfiguration system are those associated with the modes disabled by the hardware failures. The reconfiguration system will always attempt to restore disabled modes at zero disturbance and report the results to the operator. If all disabled modes cannot be restored at a zero S_a score, the operator has the option of explicitly naming the set of modes that can be relocated or of giving a level of allowable reconfiguration disturbance S_a. If S_a is given, then the reconfiguration system will first try to enlarge the solution space as much as possible, while keeping the disturbance level below S_a.

An example may help to illustrate these ideas. Suppose that we consider five modes, two associated with scanning and three with fire control. The scanning modes have individual criticality score 0.1 and joint criticality score 0.8. The fire control modes have individual criticality score 0.15 and joint criticality score 0.9. In reconfiguration, suppose that one of the scanning modes is the only mode in the failed processor and that the operator specifies $S_a = 0.6$. If we try to relocate the functioning scanning mode, then we have a disturbance score S_a given by $0.1 + 0.8 = 0.9$. The 0.1 is the individual criticality score for disturbing the functioning scanning mode. The 0.8 is the joint criticality score for disturbing both scanning modes, one of which was disturbed by the processor failure. Since $0.9 > 0.6$, we must not disturb the only functioning scanning mode in the reconfiguration activity. If we relocate two fire control modes, the disturbance score is $0.15 + 0.15 = 0.3$. If we relocate all three fire control modes, the disturbance score is $0.15 + 0.15 + 0.15 + 0.9 = 1.35$. In this example, we can only relocate the scanning mode disabled by failure plus two functioning fire controlling modes.

Generally, once the allowable disturbance level for reconfiguration S_a is given, we need an algorithm to identify the set of modes that can be moved without exceeding the specified value of S_a. It is possible that this computation has already been carried out off-line and results are stored in the database. Thus, for a given S_a, we would only need to look up the movable modes in the database. We also provide an on-line algorithm to carry out this activity. This algorithm is a modified form of the well known *first fit decreasing* (FFD) bin packing algorithm called the *reverse FFD* algorithm, which takes a set of failed modes and identifies a set of functioning modes that can be disturbed. The identified set satisfies the disturbance constraint S_a and is

Each of these formulations can be used. However, we prefer the bin packing formulation because the existence of the *first fit decreasing* (FFD) algorithm, which is known to be very fast and often produces either optimal or near optimal results [Frederickson 80, Bentley 83]. The development of our algorithm for the search of the linear subspace will be based upon the FFD algorithm.

In the single dimension case, the FFD algorithm is to first order the items in decreasing size. We then try to put the largest item into the first bin. Next,we try to put the next largest one into the first bin and so on until the first bin cannot be filled. We then use the second bin and repeat the same algorithm. If the bin sizes are different, we order the bin in increasing order according to their sizes. The idea of FFD is to get the most difficult one done first: trying to put the largest item into the smallest bin first. When the problem is multi-dimensional, we use the largest number of requirements to order items and the smallest number of its capacities to order bins. For example, if a process (item) needs 20% CPU and 10% memory then its scalar measure is 0.2. If a processor (bin) has 70% CPU available and 30% memory available for reconfiguration task, then 30% is the scalar measure.

2.4.1.2 Quadratic Phase: Minimizing Communication Traffic

When the bus communication constraints are violated, we enter the quadratic phase. In this phase, our objective is to minimize the traffic flow on the bus. Our quadratic integer assignment problem can be formulated as follows:

1. Algorithms based upon the Branch-and-Bound method [Hillier 80].

2. Algorithms based upon the estimation method [Graves 70].

3. Algorithms based upon the cluster idea used in real-time task allocation [Chu 80].

In a real-time environment, the only feasible algorithms are the clustering algorithms because either the Branch-and-Bound or the estimation method are known to be slow. The clustering algorithms are based upon the following observation. If we first group the heavily communicating tasks together before assigning them to the processors, then the solution space is significantly reduced. One can argue that this reduced space is promising because many heavily communicating processes are already bound together and they are no longer to communicate over buses.

2.4 Algorithm Development Approach

Having discussed the principles underlying the issues of disturbance and vulnerability, we now develop our algorithms. We first describe our overall two phase approach as a basis for the design of fast algorithms. We then describe the implementation of the minimization of disturbance and of vulnerability as well as the maximization of the feasibility in the context of this two phase approach.

2.4.1 The Two Phase Search Approach

We have pointed out that due to the communication constraints, our reconfiguration problem is of the type of *quadratic integer assignment problems* whose optimal solutions are generally impossible to obtain unless the problem size is extremely small. Thus, we must develop fast algorithms that focus their search in the promising area of the solution space.

The constraints of our reconfiguration problem can be divided into two classes: linear constraints such as CPU cycles and memory units and the quadratic constraints, bandwidth of the buses. Since a bus is one of the most reliable elements in the system, it is less likely to fail as compared with processor elements. This indicates that a promising area in the solution space is the linear subspace. Based upon this observation, we divide our search of the solution space into two phases: the linear phase and the quadratic phase. In the linear phase, we try to satisfy all the linear constraints. Once the linear constraints are satisfied, we have a candidate solution which will be checked if it satisfies the bus constraints. If it does, then we have found a solution. If not, we enter the quadratic phase to further minimize the communication traffic on the bus.

2.4.1.1 Linear Phase: Satisfying Processor Constraints

In the linear phase, our reconfiguration problem is of the type known as integer assignment problems. The constraints are CPU utilization, memory utilization and communication port utilization. Note that the port utilization is additive. There are generally three possible formulations to this problem.

1. Formulate the problem as optimal zero-one integer programming problem and use the branch and bound method.

2. Formulate the problem as the linear programming problem of the "cutting stock" type [Chvatal 83].

3. Formulate the problem as multi-dimensional bin packing problem [Coffman 83].

2.3.3 Trade-offs

We have identified three aspects of the cost of a particular failure: sunk costs from a failure, the present additional cost from the reconfiguration activity and the average sunk cost of next failure (the future vulnerability). We focus entirely on a single processor type, since processes in a failed processor must be moved into the same processor type. Once a failure occurs, one can compute the sunk costs by noting all the modes that have been disturbed and adding all the individual criticality scores and joint criticality scores. We label this score S_c for current score.

We now consider a particular feasible reconfiguration. We identify the modes disturbed by failure and by reconfiguration, and calculate the total disturbance score which results. This total disturbance score is denoted as S_t. The additional disturbance from reconfiguration for this solution is $S_a = S_t - S_c$. The new configuration has a future vulnerability associated with it, the average sunk costs associated with the failure of each of the processors of this type. We label this S_f for future cost. In summary, each feasible reconfiguration has two numbers associated with it, S_a and S_f.

There are two issues which need to be addressed. Ideally, one would like to find a reconfiguration in which both S_a and S_f are small. However, these two goals are often in conflict. In addition, we have discussed a typical reconfiguration. The number of such feasible reconfigurations could be enormous, far greater than could be examined in real-time. It follows that we must define heuristics which will guide us close to a solution with small values of S_a and S_f. Generally, we prefer to first specify S_a, so that the disturbance owing to reconfiguration is bounded. We then try to find a reconfiguration with a small future vulnerability S_f. The determination of an acceptable S_a requires some judgement and certainly depends on the current mission scenario. We treat the threshold limit on S_a as an input parameter to the algorithmic system. Generally, if a reconfiguration can be accomplished without further disturbance, for example, by using spares, it will be automatically carried out. If some disturbance is necessary, then input from the operator is sought. The operator, informed of the current disturbance, can either specify the set of modes which can be disturbed during reconfiguration or the maximal allowable value for S_a.

resource, while all other modes require 0 units of it. The result of this approach is that the reconfiguration algorithm will be able to avoid critical joint disturbance and avoid putting jointly critical modes in the same processor which would otherwise create an unnecessarily large future vulnerability.

The creation of pseudo resources addresses the non-linear scoring problem and allows us to keep a linear scoring approach. Notice, however, that each additional pseudo resource brings new constraints into the reconfiguration algorithm and hence reduces reconfiguration potential. Consequently, one should use this approach sparingly.

2.3.2 Future Vulnerability

When a failure occurs, certain modes will be disrupted and a criticality score can be computed using the linear rule and joint indices. After examining the impact of the failure, the system must be reconfigured, and additional disturbance may occur owing to the reconfiguration. The initial part represents *sunk cost*, the cost from the failure. The reconfiguration part can only add to the cost. Therefore, there is a powerful incentive to incur the smallest possible additional cost in reconfiguration. This is, however, shortsighted. The sunk costs which seem to be unavoidable were actually the result of the previous reconfiguration decision. The reconfiguration from the previous failure left the system in a state which lead to the current sunk cost. The current reconfiguration will create a future system vulnerability. Clearly, we would like this vulnerability to be kept as low as possible.

Now we quantify this concept. Fortunately, it is quite simple to quantify future system vulnerability using an expected value approach. We focus on a fixed processor type (e.g., UYK-44). Consider any potential configuration of processes in processors. Each processor will support one or more modes. If that processor were to fail, then the sunk cost disturbance score would be the sum of the individual mode index score plus the joint index score if any such sets are co-resident in the processor. Thus, we can compute the resulting sunk cost disturbance associated with each single processor failure. We assume that each of these processors are equally likely to fail, thus the future system vulnerability is the average of the individual total processor scores. Any configuration can be evaluated in this fashion.

2.3.1 Quantification of Disturbance

In a real-time command and control environment, a system will require a large number of modes to function properly. Clearly, some modes will be of greater importance than others. We will assume that it is possible to attach an *index of criticality* to each of the modes and that this set of indices is available to the reconfiguration algorithm. This index is a measure of the importance of one mode relative to another fixed reference mode. This index can be used as *scoring device* by which to measure the amount of disturbance caused by an interruption of a set of modes. The scoring of disturbance is a very serious issue. It is most convenient to adopt a *linear scoring* rule. This rule would operate as follows. If a set of modes ceased to function, then the total disturbance is taken to be the sum of the individual indices of criticality. This is a reasonable approach in most cases and is generally very convenient; however, there are instances where it is inadequate and even misleading. For example, suppose that we consider two modes associated with scanning: scanning mode 1 and scanning mode 2. Each may have the same index of criticality. The loss of a single scanning mode is not catastrophic, because there is a second which can fulfill part of the scanning function. If, however, both scanning modes are lost, the impact is very serious, far greater than the sum of the two index scores, because the entire scanning function has been lost. This situation is representative of a *non-linear scoring* problem. That is, the total score is not simply the sum of the individual scores.

An alternative to a linear approach is a *utility function* approach. We must define a function which takes sets of disrupted modes and evaluates them numerically. This function can be as general as the situation warrants and must be constituted by hand for all the possible failure subsets. This is a feasible, but difficult and exacting task.

We propose to adopt a hybrid, intermediate scoring approach. This approach begins with a linear scoring rule but the reconfiguration algorithm will take the non-linearities into account. Furthermore, we do this in a simple and fast algorithmic way. This approach is as follows. We consider each of the modes separately and define an index of criticality for each. Next, we consider the modes in pairs. For each pair, we determine whether the simultaneous failure of each would be of profound seriousness. If so, we create an index of joint criticality and define a pseudo resource of 1 unit for this pair. Both modes are assumed to require a 0.6 units of this pseudo resource, while all other modes require 0 units of this resource. We continue in this fashion creating critical pairs, indices of joint criticality and pseudo resources. One can continue with triplets of joint criticality indices and pseudo resources. In this case, each member of a triplet requires 0.4 units of the pseudo

remain intractable perpetually" [Karp 72]. To appreciate the difficulty, we reported the work done by Kaku and Thompson [Kaku 83]. They used one of the best known algorithms coded in Fortran on a DEC-20. It took on average 31.43 seconds to obtain the optimal solution for an 8 by 8 problem, 220.91 seconds and 1,305.65 seconds for 9 by 9 and 10 by 10 problems, respectively. The rapid exponentially increasing computation time is the characteristic of this type of problem. Owing to the very nature of this problem, we had to give up the search for optimal solutions and develop our own hybrid system of fast algorithms for the reconfiguration problem.[1]

2.3 Basic Concepts: Disturbance and Vulnerability

When a failure of a processor or other hardware element occurs, there is an immediate interruption of various functioning processes either because the processor in which they were running failed or because the communication paths being used were disrupted. It is important to realize that processes are the fundamental unit of reconfiguration; however, modes are the fundamental unit of disturbance. A system mode consists of a collection of processes. If any of those processes fails (because a processor fails or a communication path is interrupted), the mode is disrupted. Thus, modes are the unit of system functionality.

One must now reconfigure the system to overcome the failure. This can be done by rerouting messages, relocating processes, shedding load or all of these three. There are three distinct aspects to be considered:

1. The present disturbance incurred by the failure.

2. The additional disturbance caused by reconfiguration.

3. The vulnerability of the subsequent reconfigured system.

We next analyze each of these aspects. Let us begin by quantifying the disturbance.

---

[1] The essential difference between an optimal algorithm and a fast algorithm in this context is that the former must cover the entire solution space, while the latter limits the search to "promising areas" of the solution space and uses simple and fast operations. Fast algorithms can operate in real-time and generally give good solutions. However, fast algorithms cannot guarantee the optimality of their solutions.

2. Subtask A1: Dynamic System Reconfiguration

In this chapter, we summarize our work on subtask A1, best effort decision making algorithms for dynamic system reconfiguration. The objective of dynamic system reconfiguration is to re-organize a system to adapt to a changing tactical environment or to overcome hardware failures. The study of reconfiguration algorithms is important to distributed tactical systems that must operate in hostile and ever-changing environments. This work is carried out under the direction of principle investigator by Prof. John P. Lehoczky, Dr. Lui Sha and Mr. Samuel E. Shipman.

2.1 Introduction

Conceptually, the three goals of reconfiguration are minimal disturbance, minimal vulnerability and maximal feasibility, the ability to reconfigure the current task set without shedding load. The goal of minimal disturbance requires the reconfiguration algorithm not to disturb the critical modes in functioning processors, so that the system can carry out its mission without much additional disruption. The goal of maximal feasibility focuses on the restoration of as many disabled modes as possible. The goal of minimal vulnerability is introduced to ensure that as a result of reconfiguration the system is made as fault tolerant to failures as possible, so that the impacts of future system failures are lessened. Conceptually, the notions of minimal disturbance and vulnerability help to eliminate the undesirable regions in the solution space, while the notion of maximal feasibility seeks to have as large a solution space as possible.

We have successfully developed a system of algorithms that

1. minimize the disturbance to the functioning tasks in the course of reconfiguration,
2. reduce the vulnerability to future system failures,
3. and operate in real-time.

2.2 Problem Complexity and Algorithm Design Decision

In a distributed system, the reconfiguration task belongs to the class of *quadratic integer* programming problems. The term "quadratic" refers to the pairwise nature of the communication traffic and the term "integer" refers to the indivisibility of a process. The determination of the *optimal* solution to this type of problems is known to be a special type of NP-complete problem, which, except for very small sizes, "will

transaction when executing alone and follows the rules of this theory, then the following are true despite system failures:

- The consistency of shared data (database) is ensured,
- The post-condition of each transaction will be satisfied,
- Transactions can be written, modified and scheduled independently of the rest of the transactions in the system.
- The concurrency achieved by this approach is at least as great as any other consistent and correct modular approach.

1.3.2 Subtask B2: The Development of Co-operating Transactions

In the process of developing reconfiguration algorithms under Subtask A1 and transaction theory under Subtask B1, we have discovered that the current technology used for real-time executives (operating systems) is inadequate to support the realization of either of these two technologies in the context of a distributed real-time command and control system. The lack of a system level scheduling facility makes run time process level reconfiguration or a real-time transaction facility infeasible, because there is no run time mechanism to correctly assign a dispatching priority to the newly configured processor load or to the transactions that follow some concurrency control protocol. As a result of this discovery, current efforts under subtask B2 concentrate on resolving these issues by investigating real-time process management techniques.

1.4 Plans

Progress on Tasks A and B indicates a strong need to begin an immediate study of the time management problem associated with best effort decision making and transaction facilities. In the next year, emphasis will be given to the real-time aspect of distributed tactical decision making. Task A will emphasize time-driven resource management --- managing system computation and communication resources efficiently so that real-time constraints will be met. Task B will examine extension of the model of compound transactions to allow for the specification of timing constraints and development of the new theory of co-operating transactions. A more detailed discussion of CMU's plan is in Chapter 7.

5. Otherwise, run the clustering algorithm. If successful, report to the operator that the reconfiguration is successful, and exit this routine. If unsuccessful, report that the reconfiguration fails. In addition, report the modes that are disabled.

2.6.2 The Overall Reconfiguration Procedure

When a failure occurs, do the following:

1. Report to the operator the modes disabled by the failure.

2. Try to use spares if there are any. If this is successful, report to the operator that the reconfiguration is successful.

3. If this fails, run the reconfiguration routine at zero level disturbance and report the result to the operator.

4. Let the operator determine if there is any mode that can be shed and what should be the allowable disturbance threshold level, or the set of the movable modes.

5. Run the reconfiguration routine accordingly and report the result.

2.7 Conclusion

Dynamic system reconfiguration algorithms are important to the survivability of a tactical decision system that must operate in hostile and ever changing environments. We have successfully developed a system of algorithms that

1. minimize the disturbance to the functioning tasks in the course of reconfiguration,

2. reduce the vulnerability to future system failures,

3. and operate in real-time.

3. Subtask A2: Best Effort Decision Making Theory

This chapter summarizes our work on Subtask A2, best effort decision making theory. This work represents the basic research part of our effort and is primarily carried out by Prof. John P. Lehoczky.

3.1 Introduction

In a decentralized computer system, processes often function co-operatively as a team to enhance system level performance. One basic goal of the Archons project of which this research is part is to develop concepts and mechanisms which are intended to function co-operatively by using co-operative decision making. These mechanisms must function efficiently even though they will have both inaccurate and incomplete information upon which to act [Jensen 84]. In this chapter, we introduce a framework for dealing with co-operative decision making problems arising in decentralized systems in general and the Archon system specifically. Concepts drawn from queueing theory, team decision theory [Marschak 72] and information theory will be used to analyze the value of system status information for resource management. We focus specifically on the trade-offs between the completeness and the timeliness of information. Rather than treating the consensus problem abstractly, we will single out a particular problem for study: the problem of decentralized load balancing. This problem will nicely illustrate the application of one particular paradigm of consensus decision making – team decision theory. Furthermore, it illustrates the trade-offs among the cost of information (loading information at each processor), its completeness, and its timeliness. This chapter is organized as follows. Section 3.2 provides an overview of decentralized decision making Section 3.3 describes the load balancing problem and its formulation as a consensus decision making problem. Specific results characterizing the value of system status information are derived. Section 3.4 presents some ideas for further study.

3.2 Overview of Co-operative Decision Making

In decentralized computer systems, the processors often must collaborate to solve a particular system or application problem. The problems may range from those associated with the management of system resources (such as deciding which processor should process a particular task) to those arising in a specific application such as speech recognition. We propose to try to deal with these problems in a general way by casting them in a framework of statistical decision theory. Once in such a framework, it is possible to consider issues such as the importance or value of information and its decline in value with time delays.

In statistical decision theory, there is a set of possible states one of which is the "true" state. Generally, it is desired to identify the particular true state. Often data are available to assist the decision maker in identifying the true state. These data have different likelihoods for the various possible states. An example might be the load balancing problem in a distributed local computer network. The possible states represent the exact workloads at each of the processors at the time the decision is made. The data are the workload information messages exchanged by the processors. These messages are inexact and delayed in time. The particular decision made will vary in goodness with the possible states of the system.

Many decision problems arising in decentralized decision making can be cast in a statistical decision theory framework. Once in this framework, various solution techniques can be brought to bear on the problems. A particularly powerful method, Bayesian decision theory, should be considered, since it leads to optimal decision procedures. In the Bayesian formulation, the probability distribution is specified on the possible states. This is called a prior distribution. Next, we must determine the probability of the possible states with respect to observed data, which is called the likelihood function. These two components result in a posterior distribution over the possible states. The posterior distribution reflects both the prior information and the observed data. An optimal decision or action with respect to some criterion can then be determined. Furthermore, the posterior distribution can be continuously updated in real-time. Unfortunately, this program can often not be carried out in its exact form. Two fundamental difficulties arise. First, the prior distribution and likelihood function may be essentially impossible to specify. Second, in many examples the Bayesian formulation does not consider that data (for example processor status information) deteriorates in value with time. If some information is delayed, the quality of the decision made may be greatly reduced.

In many real problems, a joint likelihood function must be determined empirically. When a high dimension likelihood function is required, an empirical approach is impossible to carry out. For example, in a speech recognition context, the likelihood of a particular signal being a particular phrase will depend on many factors ranging from its context in the sentence to the sound of the actual signal. A speech recognition system will take these many diverse factors into account. The Bayesian program would call for these factors to be considered jointly rather than marginally (separately). Currently, this is an impossible task. A satisfactory but sub-optimal approach is to employ a team of specialized processors. Each processor works on a special aspect of the problem. Each processor possesses the special database and codes designed for its particular aspect of the problem. The individual processors are to function as a team of experts and are to arrive at a consensus decision. Each processor develops hypotheses and probabilities associated with the hypotheses. The team

members exchange these hypotheses and by this process enhance the marginal viewpoints to a more global view. The process must somehow converge to a consensus decision of the team perhaps by a formal method such as that of DeGroot [DeGroot 74] or by some heuristic method such as the HearSay system [Lesser 73]. The approach is somewhat similar in spirit to the "divide and conquer" approach of algorithm design but lacks the optimality of the full Bayesian program, because the joint information has been sacrificed. That is, the inter-dependence of various aspects are only approximated by the indirect approach of reaching a consensus among experts. In the following discussion, we refer to this type of co-operative decision making process as consensus decision making.

A second consideration which often leads to decentralized decision making is the delay in and the cost of interprocess communication. In many real-time applications such as load balancing, information delay often plays a dominant role, and rapid decision making may be crucial. The time delay incurred in collecting complete status information from all the relevant parties could be sufficiently long that by the time all the information is gathered and a decision is made, valuable time will have been lost, and the information itself will be substantially in error. To obtain a timely decision, the decision makers may be forced to make decisions based only on partial information. For example, in a large point-to-point network, a dynamic routing scheme is needed for message transfers through the network. Clearly, elaborate information gathering and consensus arbitration is self defeating. Instead, one might adopt decentralized procedures such as the one used in Arpanet [Kleinrock 76] and allow each node to make its own local routing decisions. Furthermore, each node will make these decisions based on only partial information, the information from nearby nodes. Information from more distant nodes will be significantly delayed and of lesser value. This illustration highlights the importance of the two concepts of timeliness and completeness of information. Since this type of co-operative decision making requires decision makers to work as a team to further the system level performance but does not require them to reach an agreed upon opinion, we will refer to this type of co-operative decision making as team decision making [Marschak 72]. A single team member is allowed to make a certain set of decisions without necessarily gaining the concurrence of the other team members. Any team member may seek information which will improve the quality of its decisions. All team members make decisions which attempt to improve overall system performance.

The consensus decision making procedure in which a single decision is reached, and the team decision procedure can be thought of as extremes on a negotiation spectrum. Generally, there are several important facets in any consensus decision making problem which will, in part, determine the most appropriate solution scheme. These include:

- Is it necessary to reach a single consensus among all decision makers? Can a single decision maker make the decision after taking the opinions and information of others into account?

- How important is the completeness of the information in making an optimal or near optimal decision? That is, how does the quality of a decision deteriorates as the completeness of information decreases?

- What is the computational complexity of the proposed decision process? Does a substantial reduction in complexity cause a major or minor reduction in performance?

- What is the total delay of a proposed decision process? This includes the information gathering, negotiation, and decision making. More importantly, how is this total delay compared with the relaxation time of the system? The relaxation time is a measure of how fast the system is changing its status.

In the following, we attempt to answer some these questions in the context of load balancing.

3.3 The Load Balancing Problem

In this section, we consider the load balancing problem in the context of a local distributed computer system such as Archons [Jensen 83] in which users create jobs at the nodes they log into. Since both the job arrival process and the required computational time for each node is unpredictable, the loads at each of the nodes will be widely variable. Some will be lightly loaded, while others will be heavily loaded. The task of the decision makers (local operating systems) is to equalize the loads in the system in order to improve the system wide performance. Balanced loads will create short queues and short response times.

Each decision maker can manage its own job queue if it chooses to process it locally. Load balancing necessitates the shifting of load from one processor to another, and this necessitates some form of consensus and negotiation involving at least a subset of the processors. At the other end of the spectrum, one processor acting as a central scheduler will assign the job to one of the processors. The no negotiation approach can be carried further by letting each of the processors schedule its own jobs. This arrangement would then be equivalent to the team decision approach whereby no consensus is needed, but the individual decision makers act to improve overall system performance. There is no a priori necessity for consensus. Rather, the degree of negotiation must be dictated by performance.

Generally, there are two aspects of load balancing. The first aspect is the matching of the structure of the data flow in the application with the architecture of the distributed computer system. This includes clustering

closely coupled processes in the same or nearby nodes, scheduling tasks with respect to precedence relations in such a way that more tasks can be executed concurrently, and matching the special resource requirements of tasks with special hardware etc. From a modelling point of view, the optimization of these aspects is typically an integer programming problem. In many dedicated real-time control systems, this is done off-line due to the lack of efficient algorithms for solving integer programming problems. A second aspect of load balancing is the control of system dynamics. In a distributed system, there is substantial redundancy built into the system. Often a class of tasks can be executed with near equal efficiency by any of a group of similar or identical processors. The problem then is the equalization of the load so as to improve the throughput and response times. In this chapter, we will concentrate on this latter aspect.

Traditionally, the load balancing is carried out by a centralized scheduler. However, this approach is inadequate for many larger scale systems which are typically loosely coupled. The inadequacy is in part due to reliability considerations and in part due to the time delay involved in routing all relevant information to a centralized location. In this chapter, we will investigate a highly parallel and highly decentralized approach. In this approach, each processor is considered to be a member of a management team, the individual actions of which are geared to increasing the system level performance. The crucial requirement is the development of an information exchange structure and an associated decision procedure which makes such an approach effective.

In summary, it appears that one reasonable approach to load balancing is the highly decentralized team approach. In this approach, each processor acts individually to increase system performance. The key ingredient is to develop an information exchange structure which will lead to increased system performance by allowing each individual processor to schedule its own jobs. The most difficult aspect lies in obtaining the relevant system status information in a timely fashion. In addition, the actions of the individual processors must be sufficiently coordinated to generate a coherent global strategy, even though it is implemented in a highly decentralized fashion. We present some results on this team formulation in the rest of this section.

3.3.1 The Value Of Perfect Information

In developing a highly decentralized team approach to the load balancing problem, an important consideration is the value of system status information. The actual information available to a processor will often be delayed, incomplete, and inaccurate. It is, however, useful to consider the ideal case of "perfect" information. System performance measures such as throughput or mean response time are influenced by unavoid-

able queueing or congestion delays and delays caused by imperfect system status information. To quantify the importance of system status information, we must separate out the extra delay incurred when information is imperfect. The approach is to measure system performance under the assumption of perfect information and to also measure it under some imperfect information scheme. The former gives an upper bound on obtainable performance, while the difference measures the loss in performance due to imperfect information.

As an illustration, let us contrast two extreme cases. At one extreme, we assume that all processors have complete, accurate, and instantaneous system information. At the other extreme, we assume that processors have no system status information at all (not even local information). In both cases, the processors will make the best possible decisions, but since different information levels are assumed, the performance achieved in the two situations will be different. The difference in performance is due solely to the availability of the status information and provides us with a quantitative measure of the value of complete information. This measure will help us to identify situations in which information is very valuable and situations where it is not.

As a specific example of this approach, we assume that the network consists of n homogeneous processors. We assume jobs arrive at the i th according to a Poisson process with mean λ_i jobs per unit time. All jobs are homogeneous and require a random amount of time to process given by an exponential distribution with mean m. We will, for the moment, ignore all other factors such as communication costs, priorities, etc. In the case of perfect information, each processor is able to act as a central scheduler. If a processor is available for work, this is assumed to be known to the other processors. No queues will form if other processors are idle. This corresponds to an M/M/n queueing system with arrival rate $\lambda = \lambda_1 + \ldots + \lambda_n$ and mean service rate 1/m. The performance of such a system can be explicitly characterized. For example, the mean response time is given by $n\rho((1-\rho)+C(n,n\rho)/n)/(\lambda(1-\rho))$ where C is the Erlang C function and $\rho = \lambda m$. The Erlang C function is bounded above by 1 and is usually small. Thus this formula can be approximated by $n\rho/\lambda$. Other quantities such as processor utilization can also be calculated.

In the case of no system status information, each processor must make a decision based only on the λ_i, m, and n. That is, only long run average information rather than dynamic loading information is available to the processors in this case. The optimal assignment scheme is probabilistic and results in an even load being put on all processors. It is important to observe that the concept of a balanced load is defined here only in the long run. All jobs are allocated so that in the long run the same number of jobs will be handled by all n processors. The lack of current and complete system status information means that the processors are not able to take current loads into account. Imbalances will result, and system performance will be decreased.

The situation of no current status information corresponds to a collection of n independent M/M/1 queueing systems with Poisson input having mean rate λ/n and mean service time m. Performance quantities are easily calculated for such a system. For example, the mean response time is given by $n\rho/(\lambda(1-\rho))$. It is interesting to compare the mean response time in these two extreme cases, the complete information case and the no information case. The two differ by the multiplicative factor $1-\rho + C(n,n\rho)/n$. One might refer to this quantity as the "information factor". The factor is equal to 1, its minimum value, when $\rho=0$, and it decreases to 1/n as ρ increases to 1. This means that at low traffic intensities (ρ near 0), system status information provides no significant reduction. This factor can be calculated for any traffic intensity ρ and shows that information is very valuable at moderate and high intensities. Furthermore, the value of information increases with the number of processors in the network.

3.3.2 The Value Of Local Information

In this section, we continue to assume a situation in which processes and processors are homogeneous. Furthermore, we do not consider the communication delays or the time cost of transferring tasks in this section. Now we wish to study the situation in which each processor has only local information (the status of its own job queue) and knows the long run arrival rate of work. The arrivals are assumed to be independent Poisson processes with common parameter λ, while service times are again exponential with mean m. Each processor upon receiving a job must determine whether to process it locally or send it to another processor. This decision must be based only on the local queue length and is made without any negotiation. We also assume that if it is sent to another processor, that processor must accept the job and cannot send it further. This assumption is useful in that it prevents excessive job swapping, and it simplifies the analysis.

A general class of control policies within which an optimal policy must lie consists of a set of probabilities, $\{p_n\}$, where p_n gives the probability that the processor accepts a newly arrived task when its local queue consists of n other tasks. These probabilities must be decreasing in n. Optimal stochastic control theory suggests that the optimal p_ns will be non-randomized (either 0 or 1). This reduces consideration to a 'control limit' policy: accept a task if and only if the local queue, including the one being served, consists of L or fewer tasks, otherwise send it to another processor chosen from among the other candidates. The best value of L will depend on the traffic intensity. If the traffic intensity is small, then $L = 1$ is appropriate. For larger values of ρ, L must be larger. It remains to determine the optimum value of L for each ρ, and to assess the performance of the resulting local information system.

Once L has been determined, there is a second phase to the policy - the choice of the processor to send the unaccepted task. A number of policies, both deterministic and stochastic can be used. An optimal policy must equalize the load on the processors in the long run. Beyond that requirement, we expect that the optimal policy will minimize the coefficient of variation of the resulting arrival processes. The form of the optimal policy is still an open question. We choose a policy in which each processor selects randomly from the remaining $(n-1)$ available choices.

The L policy queueing system has not been studied before, and it is quite complicated to produce exact analytic solutions. Fortunately, a simple approximation can be used which is very accurate for large values of $n(1-\rho)$. We treat each of the n processors as independent birth-death processes with birth rate $b_n=\lambda(1+s)$ for $n \leq L$, and $b_n=\lambda s$ for $n>L$, while the death rate is $1/m$ for all states. The parameter s is used to connect the queues together and is chosen to achieve equilibrium. The unknown s is a root of a polynomial equation. Once s has been determined, approximate equilibrium distribution and its mean can be found. The derivation is given in the appendix along with simulation results which attest the high accuracy of the approximation. If we let $F=n\rho/(\lambda(1-\rho))$, then we can determine the mean response times to be given by

POLICY	MEAN RESPONSE TIME
no information	$F[1]$
local control, L=1	$F[1/(1+\rho)]$
local control, L=2	$F[1-\rho+\rho/((1+\rho)^2)]$
perfect information	$F[1-\rho+C(n,n\rho)/n] \sim F[1-\rho]$

The results are rather surprising. The percentage reduction in mean response time for perfect information over no information is ρ. The percentage reduction for the L=1 policy is $\rho/(1+\rho)$. Of the total ρ percent reduction for perfection, $1/(1+\rho)$ of it can be obtained using only local information. Even at high intensities, local information gives over 50% of the possible gain. If one allows general L's, then at least 62% is possible for any traffic intensity! It is worthwhile to point out that some non-local information can be obtained at nearly no additional cost by keeping track of the source of each task in the local queue. Hence one can further improve system performance by using this "free" non-local information. For example, if a processor decides to ship out a task, it should be sent to a processor which has not shipped it anything in its current queue. This all suggests that very effective control can be exercised at low traffic intensities using just local control. At higher intensities, the mean response times become quite long. Even though local control can capture a high percentage of the overall gain possible from perfect information, it becomes important to refine the decision making with extra information. Specifically, one needs to make a more informative choice

of the processor selected to process the task. The purely random choice described earlier is inadequate. Several such schemes are suggested in the next section.

3.3.3 Approaches to Decentralized Load Balancing

Once one has introduced a measure of the value of information, it remains to develop an algorithm which will nearly achieve the theoretical upper bound on performance. In general, processors will send messages to inform other processors of their status. The more frequent the messages, the more information other processors will have. Unfortunately, these messages create overhead which serves to degrade system performance. To deal with this tradeoff, one can draw an analogy with team decision theory. Information is valuable and worth the cost only if it causes the recipient to change his decision and results in a lower overall cost. A message should be sent only if it has a sufficiently large information content to justify the cost of sending it. Here we are ignoring the reliability considerations which may dictate that a message be sent periodically. We use the word 'information' in the Shannon sense. The information gained from a message is equal to the uncertainty removed. Only messages describing relatively unusual or surprising system states should be transmitted. In the load balancing context, the high information content messages are those which describe overloading or underloading conditions. This suggests that processors should transmit status messages according to a control limit policy: either when it is underloaded or overloaded relative to its equilibrium distribution. The exact control limits are determined to optimize the tradeoff between the cost of information and the gain in performance from it.

The control limit policy described in the previous paragraph should provide a very efficient decentralized control algorithm. The policy is, however, one based on long run equilibrium calculations which ignore short run fluctuations. A processor will send a status message when its workload deviates from its long run expectations. It is possible that there could be a heavy influx of jobs over a short period (or a very small influx), and many processors will have above (below) average workloads. This causes many status messages to be sent which further degrades the system. This effect can be overcome by having control limits which are determined dynamically and change based on short term load fluctuations. Unfortunately, this would seem to necessitate sending even more messages to identify these short run fluctuations. There is, however, an alternative approach. This approach is based on having each processor construct a system status model. If we assume that the processors are linked by a bus connection, then each processor must do intensive bus monitoring. Each processor must keep track of all traffic on the bus, noting its source, its destination, and if

possible its estimated processing time. This information allows each processor to build and update a system status model. Such a model would predict the current workload at each processor. Based on this model, the processor could act as a central scheduler to determine the best processor to handle any particular task. The quality of these decisions will be totally a function of the accuracy of the model used. At first glance, it would seem that the system status model might be accurate for a short time period, but its quality would quickly deteriorate as time passes, due in part to the uncertainty of the exact processing requirements of any particular task. It is necessary to introduce some sort of feedback control to keep the model accurate. This can be also done in a highly decentralized efficient manner. Under the assumption that the processors are coupled by a bus, they will all see the same traffic. As a result, they will all build identical system status models. Thus the models used by each of the processors gives essentially identical predictions of the workload at any particular processor. In addition, each of the processors has extra information in that it knows the exact workload for itself. Each processor can compare its own workload with the workload as predicted by the common system status model. The two will of course deviate. It is only important to notify the other processors when this deviation becomes significant. This is again done on a control limit basis. When the actual load is significantly larger or smaller than the load predicted by the model, that processor must send a message to all other processors to have that part of the model updated. The exact control limits must be determined to tradeoff the increase in performance with the overhead costs in sending the message. In this fashion an effective feedback control mechanism can be established in a highly decentralized fashion.

3.4 Future Research

The previous section indicates how one can quantify the value of system status information. We wish to generalize these results to a broader context. This will be done as follows. We will assume that both jobs and processors may be heterogeneous. Further we will reintroduce the costs (in time) of communication associated with sending jobs around the network. A number of models are possible to handle this greater degree of generality. We might assume that processes arriving at processor i (at rate λ_i) can be processed by any other processor, but the amounts of time required will vary. A number of factors will cause these time differences: differing communication times, the capabilities of the particular processors, the requirements of the job, the location of the data, etc. We assume that the time to process a job at processor j is random and has a distribution F_{ij} with mean m_{ij} and variance s_{ij}. From this structure, one must evaluate the system performance under a full information and a no information structure.

Under the assumption of no dynamic information, the decisions must be based solely on the parameters λ_i and F_{ij}. Again a probabilistic algorithm will be optimal. Performance quantities such as the mean response time can be calculated from the Pollaczek-Khinchin formula. The full information optimal performance evaluation can be carried out by a straightforward Lagrange multipliers argument.

It is intuitively clear that heterogeneity in general reduces the value of system status information. This observation follows because knowledge of the availability of a particular processor will be useless for jobs which are mismatched to that processor. Heterogeneity increases the number of such mismatches and thus reduces the number of cases where information is of value.

In summary, there remain a variety of important issues yet to be investigated. First, there is a need to evaluate the response times of systems more general than the one analyzed in sections 3.1 and 3.2. This is outlined above. Second, one must evaluate the performance of the various control strategies outlined in section 3.3. These initial results do, however, indicate that the team decision approach with limited local information will offer a very high performance decentralized system.

3.5 Appendix

The behavior of the network under an L policy can be determined approximately using a simple birth and death process model. We focus attention onto a single node. This node receives external input according to a Poisson (λ) process. It accepts and queues these tasks if there are currently fewer than L tasks there. It sends the work elsewhere if it currently has L or more tasks. The node also accepts tasks sent by other nodes that invoked the L policy. These tasks must be kept and processed. That is, each task can be only sent once by a node. We assume the same external input rate λ and exponential service rate $1/m$ at each node.

The queue length process at each node is treated as a birth and death process with constant death rate $1/m$. The birth rate is $b_i = \lambda + \lambda\theta = \lambda(1+\theta)$ if $0 \leq i < L$ and is $\lambda\theta$ if $i \geq L$. Here θ is the fraction of traffic forwarded by a node and is $\sum_{j=L}^{\infty} \pi_j$, where $\{\pi_k\}_{k=0}^{\infty}$ is the equilibrium distribution of the birth-death process. One can find the equilibrium distribution from standard birth and death theory.

$$\pi_k = \pi_0 \prod_{i=0}^{k-1} (b_i m_{i+1}), \quad 1 \leq k < \infty$$

$$\sum_{k=0}^{\infty} \pi_k = 1.$$

Consequently,

$$\pi_k = \begin{cases} \rho^k(1+\theta)^k & 1 \leq k \leq L \\ \rho^k \theta^{k-L}(1+\theta)^L & k > L \end{cases}$$

3.5.1 Case 1: L = 1

We consider the case $L = 1$. Here, the equilibrium distribution is given by

$$\pi_k = \begin{cases} \rho(1+\theta)\pi_0 & k=1 \\ \rho^k \theta^{k-1}(1+\theta)\pi_0 & k>1 \end{cases}$$

where $\theta = 1 - \pi_0$, $1 = \sum_{j=0}^{\infty} \pi_j$.

The condition $1 = \sum_{j=0}^{\infty} \pi_j$ yields $\pi_0(1+\rho)/(1-\rho\theta) = 1$. The further requirement $\theta = 1 - \pi_0$ gives $\pi_0 = 1 - \rho$, so $\theta = \rho$. This gives the equilibrium distribution in terms of ρ alone,

$$\pi_k = \begin{cases} 1-\rho & k=0 \\ \rho^{2k-1}(1+\rho)(1-\rho) & k \geq 1 \end{cases}$$

The mean queue length can be calculated directly from this distribution to be $\rho^2/[(1-\rho)^2(1+\rho)]$. Little's formula can be used to find the waiting time = $F(1/(1+\rho))$.

3.5.2 Case 2: L = 2

The analysis for L = 2 is similar to the L = 1 case. The equilibrium distribution is given by

$$\pi_k = \begin{cases} \rho(1+\theta)\pi_0 & k=1 \\ \rho^k\theta^{k-2}(1+\theta)^2\pi_0 & k\geq 2 \end{cases}$$

where $\sum_{i=0}^{\infty}\pi_i = 1$, $\theta = 1-\pi_0-\pi_1$.

The three conditions, $\sum_{i=0}^{\infty}\pi_i=1$, $\theta = 1-\pi_0\pi_1$ and $\pi_1 = \rho(1+\theta)\pi_0$ can be used to find the equilibrium distribution. It is $\pi_0 = 1-\rho$ and

$$\pi_k = \begin{cases} \rho(1+\rho)(1-\rho)/(1+\rho(1-\rho)) & k=1 \\ [\rho^3/(1+\rho(1-\rho)]^k(1+\rho)^2(1-\rho)/\rho^4 & k\geq 2 \end{cases}$$

Once again, the equilibrium queue length can be found this distribution. The waiting time is given by $F(1-\rho+\rho/(1+\rho)^2)$

3.5.3 Simulation Results

It is important to determine the accuracy of the previous approximation. A simulation was carried out for that purpose. Selected results are presented below for the case of 5 nodes. The results show that the approximation is extremely accurate for traffic intensities up to 0.5 and still accurate when traffic intensity is 0.7. At high traffic intensities, the network becomes difficult to simulate accurately as extremely long runs are needed. The approximation will work better as the number of nodes increases, but simulations are difficult to carry out in such cases. The followings are simulation results and theoretical results of the probability of n jobs in the queue. In addition, the mean queue length of simulation results and theoretical results are also compared.

Traditionally, studies on concurrency control have focused on the issue of database consistency, rather than transaction correctness. Under a non-serializable schedule, transactions cannot be considered as if they are executing alone. It is, therefore, unclear if a transaction which satisfies its post-condition when executing alone will still satisfy its post-condition when it is run non-serializably, even if the non-serializable schedule is consistent. Although many proposed non-serializable control methods are shown to be powerful tools to solve various specific application problems, non-serializable concurrency control methods will be difficult to use in a general transaction facility until the problems of the consistency of the database, the correctness of transactions and the modularity of scheduling transactions are coherently solved.

In this chapter, we develop a theory of modular concurrency rules – a theory of provably consistent and correct concurrency control methods that allow one to write, modify and schedule one's transactions independently of the others. Our theory is a generalization of serializability theory and provides at least as much concurrency as any other modular scheduling method. Before the formal investigation, we give an informal overview of our theory. From a programmer's point of view, our work is a formal theory for decomposing a transaction system. Under serializability theory each transaction is treated an atomic unit of operations and the database is treated as a non-divisible unit of data objects. In our approach, both the database and transactions are decomposed as follows.

1. The database is partitioned into consistency preserving *atomic data sets*. As long as the consistency constraints of each atomic data set are satisfied individually, the consistency of the database is maintained. However, data objects in different atomic data sets need not be independent or unrelated, and data objects can be added to or deleted from any of the dynamic data structures (e.g. linked lists) in an atomic data set.

2. Each transaction is independently partitioned into a partially ordered set of correctness preserving *elementary transactions*. As long as the post-condition of each executed elementary transaction is individually satisfied, the post-conditions of the partitioned transaction (called a *compound transaction*) are also satisfied.

   An elementary transaction can have the structure of nested transactions, and it preserves the consistency of the accessed atomic data sets. However, elementary transactions in a compound transaction need not be independent computational units. That is, the results of computations can be exchanged by the elementary transactions of the same compound transaction.

Having divided the database and transactions, we next employ some concurrency control protocol to ensure that the elementary transactions of each of the compound transactions are executed serializably on a atomic data set by set basis. The protocol defined in this chapter ensures that:

5. Subtask B1-1: Atomic Transaction Theory: Modular Concurrency Control

In this chapter, we summarize our work on the concurrency control part of atomic transaction theory, which is part of Subtask B1. Concurrency control addresses the problem of how to run a set of transactions in a distributed system consistently and correctly with a high degree of concurrency. This work is primarily conducted by Dr. Lui Sha and Prof. John P. Lehoczky.

5.1 Introduction

Serializability theory has been widely accepted as the basis for concurrency control, because it ensures the consistency and correctness of concurrency control. Under serializable schedules, the results of executing a set of transactions will satisfy both the consistency constraints of the database and the post-conditions of transactions as long as each individual transaction is consistent and correct when executing alone. Furthermore, with serializable schedules one can write, modify and schedule[6] any single transaction independent of any knowledge of the rest of the transactions in the system. Such a modular property of concurrency control is very attractive in the development of a general purpose transaction facility in which transactions are frequently modified. We believe that the properties of consistency, correctness and modularity account for both the popularity of serializability theory and the continuing interest in the study of various protocols that support the serializability of concurrency control [Weihl 84, Attar 84, Papadimitriou 84, Mohan 85].

On the other hand, there are many proposed non-serializable scheduling methods that can provide a higher degree of concurrency than that available using serializable schedules [Allchin 82, Schwarz 82, Garcia-Molina 83, Lynch 83, Schwarz 84]. Since serializability theory is optimal when only transaction syntax information is used for scheduling transactions [Kung 79], many proposed non-serializable scheduling approaches have typically focused on various methods of using transaction semantic information to maximize system concurrency. This approach can lead to significant increases in concurrency. There is, however, a potential accompanying disadvantage. When a set of transactions are jointly scheduled through the use of semantic information, the modification of even a single step of a single transaction could result in the rescheduling of the rest of transactions. There is a second problem which arises with the use of non-serializable schedules.

---

[6] For example, one can use the two phase lock protocol to schedule one's transaction [Eswaran 76].

- Use *hard* **deadline scheduling** -- Use all value functions with the same constant value prior to their deadlines, 0 after the deadlines.

- **Maximize the number of processes scheduled in a given time interval** -- Use all value functions with the same constant value both before and after their deadlines.

- **Use user-defined priority scheduling** -- Use value functions with constant values, but whose values represent their priorities.

- **Minimize the average lateness** -- Use value functions which are constant prior to the process deadline, but linearly decreasing after the deadline.

A solution for this problem must produce (at least) the following actions in some form:

- Continuously refine the estimate of the computation time for each process.

- Determine the probability that an overload condition exists such that one or more currently schedulable processes will miss its deadline.

- Determine the best execution sequence of the currently known schedulable processes.

- Initiate the shedding of load if an overload condition currently exists and has a high probability of continuing.

function of process p_i drops below some user-defined value, the scheduler may decide that there is insufficient value to the system in continuing its execution, and cause its abortion. This action will (perhaps slowly) purge the system of processes erroneously capturing a disproportionately large portion of the system resources. The type of functions definable for V_i will determine the actual scheduling performance of the system.

Each scheduling computation results in the determination of a process ordering $(m_1, \cdots, m_n)$, where p_{m_j} is the j^{th} process to be scheduled. Scheduling will be considered optimal if, with respect to the available information at the time of the scheduling decision, β is maximized, where $\beta = \sum V_i(T_i)$ and T_i is the expected completion time of p_i using the scheduling sequence currently decided upon (if p_i is the j^{th} process to be scheduled, then $T_i = \sum_{k=1}^{j} C_{m_k}$).

It is evident that no priority has been defined for p_i. The use of the value function renders such a concept unnecessary for this research, since the relative priority of all competing processes, at any point in time, will be determined by their respective value functions at their estimated completion time. Thus their priorities will vary, but will always reflect the estimated likelihood of completing prior to their deadlines. A separate question is how an application will define the relative importance of a number of competing processes, but this question will not be considered in this research. A number of potential solutions to this problem exist, and their choice will depend on the behavior desired as well as the actual type of value functions used.

This model encompasses both periodic and non-periodic processes in that any individual execution of a periodic process can be described as shown above in exactly the same way as for a non-periodic process. The only difference is that the periodic process immediately becomes schedulable when it terminates, since its next request time is already known. We can beg the question of overlapping executions since, with our model, they can either be avoided (by ensuring that a periodic process' value function drops to zero prior to its next request time), or handled by allowing multiple instances of a process to be simultaneously schedulable (assuming that processes are reentrant, and that the average load does not become larger than the processor capacity).

It should be noted that the use of the arbitrary value function in this computational model subsumes a number of scheduling policies which are commonly used or desirable in determining scheduling algorithms such as:

A schedulable process, p_i, is one which is not currently awaiting any external[5] event in order to be scheduled, and which has an outstanding request for processing time. We define a process as schedulable as soon as it has met this requirement, and it remains schedulable until one of the following conditions has occured:

- Its process defined deadline event has been signaled and the process has blocked waiting for some future external event.
- It has terminated.
- It has been aborted by the system.

Thus, the request time R_i for a schedulable process may be either a future or a past time. If the request time R_i is a future time, the process is not currently a candidate for dispatch, but its attributes may be considered in the current computations of load from which current scheduling decisions are made. The reason for this will become clearer as we progress with some of the scheduling algorithms and heuristics for this decision. Process p_i cannot be dispatched prior to its request time.

The computation time C_i is a stochastically defined value representing the expected time to process p_i (estimated time remaining if p_i has already begun processing), not including non-requested system overhead or preemptions. The source of this value is expected to be an actual measurement by the system itself. Other possible sources are the process programmer at implementation time or a predicted value using process information combined with system measurement. The choice of source(s) for this value is one of the questions to be considered as part of the policy/mechanism discussion. In this research, the distribution of C_i is one of the issues to be discussed.

The deadline D_i is a time provided by the requesting process at the time R_i is defined, making p_i schedulable. The importance of the deadline is determined by the value function V_i; in fact, the deadline itself, without the value function, has no particular importance. V_i, defined in this research as a non-negative function, defines the value to the system for completing p_i prior to D_i. Its value is used by the scheduling function to determine the best sequence in which to schedule each of the available processes. If the value

---

[5] An external event is an event which is not under control of the process. Examples of such events are I/O completion, asynchronous requests from another process, and timer countdown.

Following the definition of the computational model, the set of possible scheduling actions must be defined. This includes the scheduling decisions as well as the determination of an overload condition in which it is apparent that, at a given processor, the resources are inadequate to meet some deadlines. If resources are inadequate to meet some of the deadlines, a user policy must be defined to specify how the system should respond. A set of potential policies for such cases will be described, and mechanisms (algorithms) to implement these policies will be defined.

4.3.2 Deadline Scheduling Model

Consider some small number n of processors (say 4), in a fully connected distributed system[4].

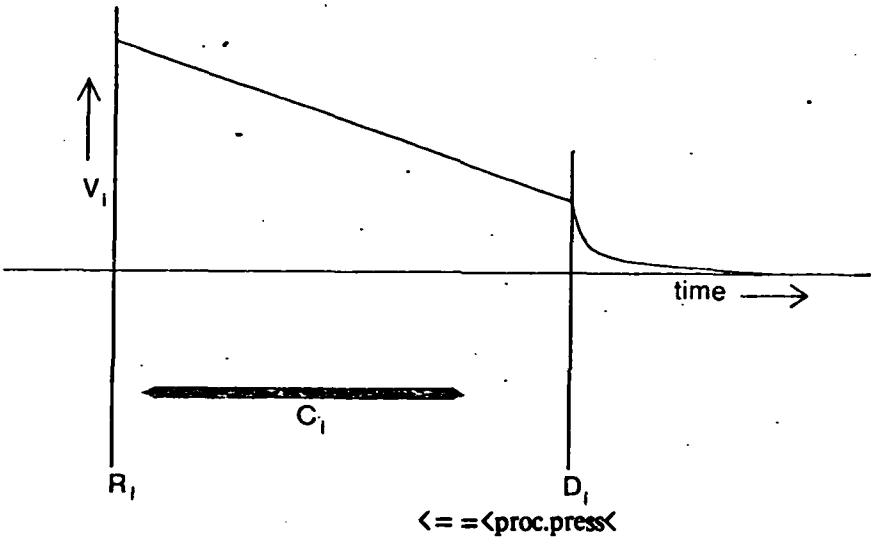


Figure 4-1: Process Model Attributes for Process i

The model for this problem consists of a set of schedulable processes p_i resident in each processor. Each such process has a request time R_i, a deadline D_i, an estimated computation interval C_i, and a value function $V_i(t)$, where t is a time for which the value is to be determined. Figure 4-1 illustrates these process attributes for a process with a linearly decreasing value function prior to its deadline, and an exponential value decay following its deadline. The illustration depicts a process which is dispatched after its request time and which completes prior to its deadline.

---

[4]We will use a small number of processors for convenience in performing simulation of the resulting algorithms, but the algorithms themselves will contain no assumptions regarding the actual number of processors.

4.2.5 Distributed Decision Making

The *Byzantine Generals Problem* [Lamport 82] [Dolev 82] [Lynch 82] is the name given to the problem of reaching a consensus on a decision in an environment in which some of the decision makers have failed (or may even be antagonistic to the making of a correct decision). It has been proved that in a fully connected network of decision makers [Lamport 82], a correct decision can be effected as long as more than two thirds of the decision makers have not failed. The algorithms presented, however, while guaranteed to produce a correct solution, require full synchronization of all decision makers, and require a large number of messages. It should be noted that in a completely asynchronous system (i.e., one with no complete event ordering such as a clock), no Byzantine agreement is possible in the presence of any failure [Fischer 82]

4.3 Hard-Real-Time Scheduling

Making a best-effort decision in a distributed system can be accomplished using a number of techniques spanning several computer science disciplines. There are no specific bounds on the techniques employed, so algorithms will be developed using ideas from such areas as artificial intelligence, decision theory, and non-monotonic logic, as well as combinations of these. It is expected that each of these techniques will have a number of positive and negative characteristics in the context of decentralized operating system resource management.

This research will be performed in four partially overlapping phases:

1. Problem Specification
2. Algorithm Development and Analysis
3. Experimentation
4. Evaluation

4.3.1 Problem Specification

First, a detailed statement of the problem to be solved must be produced, defining an instance of a distributed system for which deadline management can be described precisely and simply, as well as the available information from which to make the decisions. To define such a problem, it is necessary to describe the computational model of the processes to be scheduled and for which algorithms will be designed.

whether to plan a picnic for Saturday). The decision maker first constructs a matrix with each row for a potential decision and each column for a potential relevant state, placing in each matrix element a value of the utility or desirability of having made that decision if the corresponding state occured. Then the decision maker constructs a similar matrix containing the probability that that state will actually occur (possibly conditional on the decision to be made). The row-wise sum of the matrix produced by multiplying each element of the utility matrix by the corresponding element of the probability matrix defines a vector representing the optimal ordering of the set of potential decisions.

4.2.4 Real-Time Deadline Management

Research into the general scheduling problem has progressed for a long time as a part of operations research, where it has found application in the scheduling of such activities as manufacturing production. Graves [Graves 81] has provided an excellent synopsis of this background, including a taxonomy of production scheduling problems. He identifies four levels of complexity with respect to scheduling decisions; tractable solutions have been identified for only the simplest of these levels while the rest have been proved to be NP-complete or NP-hard.

In particular, the level of complexity most similar to the problem to be considered in this research, soft-real-time deadline scheduling with constant value functions for met deadlines, has been proved to be NP-complete [Karp 72]. Algorithms for this problem have been described [Sahni 76] both for optimum scheduling (obviously, in exponential time) and for a heuristic approach for which an optimal solution can be missed by a determinable factor in $O(n^3)$ time. The model which we are using to define the scheduling problem to be handled includes this problem as a special case.

In a classic paper, Liu and Layland [Liu 73] show that the simple deadline scheduling problem with a set of independent periodic processes whose computation times are exactly known and identical for every period, whose deadlines are equal to their period, and which are running in a single processor, is solvable with a simple algorithm, and that the schedulability of such a simple system is easily determinable in advance. While this result is interesting, the cases covered are much too unrealistic to be directly applicable to our problem.

applying a best effort to the real-time process scheduling problem, particularly in the areas of data understanding and knowledge and inference rule representation.

4.2.2 Non-monotonic Logic

Decision making is dependent on a determination of the state of the relevant world as assessed by the decision maker. The decision to be made can be viewed as a consequence (deduction) from this perceived state, which can be represented as a set of predicates. Normal formal logic, from which inference rules would be taken to produce such deductions, can be described as *monotonic* in the sense that the addition of any new axiom (i.e., an additional observation of state) which is not in conflict with existing axioms, can never invalidate a previously drawn conclusion. In a situation characterized by a large quantity of incomplete or inaccurate information, such deductions may later be invalidated by the presence of new information Logic in which inferences may be drawn using such data is called non-monotonic logic [McDermott 80] [McDermott 82], and is generally characterized by the use of a logical operator meaning *"is consistent with"* on conclusions drawn.

In another form of monotonic logic, inferences which can be described only as "consistent" with perceived state information can be handled by the use of *fuzzy* sets. Normal set theory deals with sets whose boundaries are clear-cut; an element is either a member of a given set or it is not. Members of fuzzy sets [Kaufmann 75] have an associated membership value which determines the degree of membership of that member in the set. Fuzzy logic refers to the manipulation of predicates with fuzzy implications (e.g., High value processes should be executed early). Discussions of such logic inference rules are given in [Zadeh 79].

4.2.3 Decision Theory

There are several mathematical techniques developed to construct decisions based on the opinions (inexact measurements) of a number of co-decision makers. Stankovic [Stankovic 83] describes a heuristic for job assignments in a decentralized, distributed environment using Bayesian decision theory [Jeffrey 84], while DeGroot describes iterative solutions given a matrix of opinion weights and a vector of opinions [DeGroot 74].

Bayesian decision theory may be used in an environment in which a decision must be made whose value is determinable only when evaluated in the light of an unpredictable future global state (such as determining

Questions of policy/mechanism separation [Wulf 81] are also related to this problem. If one or more deadlines must be missed, which deadlines should be selected? Clearly a policy decision must be made, and the range of potential policies for this decision will be determined, with the required mechanisms defined, implemented, and evaluated. It is these mechanisms which will reflect the best effort approach, and their support of the potential policies will be analyzed.

4.2 Related Research

This area of research is the combination and extension of a number of previous efforts. As a result, there are several areas of research which can be considered related to this work. Here, we will describe research in five related areas:

- *Artificial Intelligence* -- particularly techniques for knowledge representation and rules of inference related to decision making (planning).
- *Non-monotonic Logic* -- studies related to the handling of multi-valued logic, fuzzy logic (truth assertions with respect to fuzzy sets), and associated truth determination.
- *Decision Theory* -- work related to the process of combining sets of observed states of nature, the probabilities associated with these states in the event of a given set of decisions, and the ordering of possible decisions based on this information (e.g., Bayesian theory).
- *Real-Time Deadline Management* -- research on deadline scheduling in a real-time system.
- *Distributed Decision Making* -- efforts toward the problem of reaching a consensus in a distributed system in the presence of failures.

4.2.1 Artificial Intelligence

Little or no work has been performed in the application of knowledge collection, knowledge representation, and the resulting inferences to the problem of scheduling, load balancing, or process reconfiguration. It seems that these areas could be critical to the efficient handling of these decisions. Stefik [Stefik 82] delineates a number of approaches to problems involving expert systems, outlining the primary techniques used in developing them based on the overall system type (e.g., planning, diagnostic, etc.). This problem shares a number of the aspects of planning problems, in that the solution must evaluate the data determining its quality and meaning, search the space of possible scheduling decisions estimating the result of each evaluated decision, and make a decision which is adequately close to optimal. It is expected that at least some of the techniques involved in the design of expert planning systems will be applicable to the problem of

4. Subtask A3: Multi-Processor Real-Time Scheduling

The chapter summarizes Subtask A3, our recently initiated work on multi-processor real-time scheduling. In a modern combat environment, making and carrying out the decision in time can be as important as the quality of the decision itself. Missing the deadlines could mean the failure of an otherwise successful operation. In this chapter, we describe our initial work on a value function based approach for real-time multi-processor scheduling. This work is primarily carried out by Mr. Douglass Locke as part of his PhD research.

4.1 Introduction

An operating system managing a real-time application shares a large number of functions with a normal timesharing operating system, but differs most significantly in one principal area -- the management of time deadlines. The typical time sharing operating system manages a number of independent applications, promoting some concept of fairness (defined by the system administrator) among them. In the real-time system, a single application[3] uses the system resources to solve a single problem, and includes a set of deadlines which must be met in order to satisfy its specifications.

The research which we undertake here is to study deadline management in a multi-processor environment, in which the time allocation decisions must be made using a best effort approach. In fact, for this problem a best effort approach must be used because of the built-in uncertainties even if complete and accurate global system state information were available, because of the stochastic nature of the information available about schedulable processes and the limited decision making time available in a real-time system.

A considerable amount of research has been done when deadlines could be met, but relatively little information is available about scheduling decisions when available resource limitations require that one or more deadlines cannot be met. Our approach will be designed to maximize the value of the available state information to make the deadline scheduling decisions, particularly in those cases where deadlines cannot be met.

---

[3] A single application means a set of processes working together and sharing resources toward a common goal. While it is possible for more than one such application to run concurrently in a real-time system, it is usually true that the real-time commitment of the system will be made only to one; any others will execute as background applications.

Run 8: Traffic Intensity = 0.7, Policy: L = 2

	Simulation	Theory
Prob[n=0]	0.301	0.300
Prob[n=1]	0.279	0.295
Prob[n=2]	0.273	0.290
Prob[n=3]	0.086	0.082
Prob[n=4]	0.034	0.023
Prob[n=5]	0.015	0.007
Prob[n=6]	0.007	0.002
Prob[n≥7]	0.005	0.001
Mean queue length	1.371	1.267

Run 5: Traffic Intensity = 0.5, Policy: L =1

	Simulation	Theory
Prob[n=0]	0.502	0.500
Prob[n=1]	0.365	0.375
Prob[n=2]	0.094	0.094
Prob[n=3]	0.027	0.023
Prob[n=4]	0.008	0.006
Prob[n=5]	0.003	0.002
Prob[n≥6]	0.001	0.000
Mean queue length	0.687	0.666

Run 6: Traffic Intensity = 0.5, Policy: L = 2

	Simulation	Theory
Prob[n=0]	0.502	0.500
Prob[n=1]	0.292	0.300
Prob[n=2]	0.179	0.180
Prob[n=3]	0.022	0.018
Prob[n=4]	0.004	0.002
Prob[n≥5]	0.001	0.000
Mean queue length	0.737	0.722

Run 7: Traffic Intensity = 0.7, Policy: L = 1

	Simulation	Theory
Prob[n=0]	0.299	0.300
Prob[n=1]	0.344	0.357
Prob[n=2]	0.167	0.175
Prob[n=3]	0.087	0.086
Prob[n=4]	0.046	0.042
Prob[n=5]	0.025	0.021
Prob[n=6]	0.014	0.010
Prob[n=7]	0.008	0.005
Prob[n≥8]	0.010	0.004
Mean queue length	1.468	1.365

Run 1: Traffic Intensity = 0.1, Policy: L = 1

	Simulation	Theory
Prob[n=0]	0.902	0.900
Prob[n=1]	0.097	0.099
Prob[n=2]	0.001	0.001
Prob[n≥3]	0.000	0.000
Mean queue length	0.099	0.101

Run 2: Traffic Intensity = 0.1, Policy: L = 2

	Simulation	Theory
Prob[n=0]	0.901	0.900
Prob[n=1]	0.090	0.091
Prob[n=2]	0.009	0.009
Prob[n≥3]	0.000	0.000
Mean queue length	0.108	0.109

Run 3: Traffic Intensity = 0.3, Policy: L = 1

	Simulation	Theory
Prob[n=0]	0.702	0.700
Prob[n=1]	0.268	0.273
Prob[n=2]	0.026	0.025
Prob[n=3]	0.003	0.002
Prob[n≥4]	0.001	0.000
Mean queue length	0.333	0.329

Run 4: Traffic Intensity = 0.3, Policy: L = 2

	Simulation	Theory
Prob[n=0]	0.699	0.700
Prob[n=1]	0.225	0.226
Prob[n=2]	0.074	0.073
Prob[n=3]	0.002	0.001
Prob[n≥4]	0.000	0.000
Mean queue length	0.379	0.375

1. The database consistency constraints will be satisfied.

2. The post-condition of each (compound) transaction will be satisfied.

3. The transaction scheduling approach is modular and provides at least as much concurrency as any other consistent and correct modular concurrency control method.

The concepts of modularity and atomic data sets formally defined in this chapter have intuitive meanings. A modular concurrency control method is one which allows a programmer to write, modify and schedule his transaction independently of other transactions. For example, serializability theory is a modular scheduling method, because a programmer can write, modify and then schedule (e.g. using the two phase lock protocol) his transaction independently of other transactions in the system. In other words, a modular concurrency control method ensures that any particular written and scheduled transaction need not be changed, when other transactions are written, modified or rescheduled.

We assume throughout that the consistency constraints of the database are fixed while transactions are being added or modified. It is, of course, possible that the specification of database consistency constraints could be changed during the development of the transaction system and this could in turn force the transactions to be rewritten and rescheduled because transactions must maintain the database consistency constraints. However, the task will be easier if a modular approach is used, because one can examine the impact of new consistency constraints on each of the transactions *individually* and perform the necessary modification *individually*.

Intuitively, an atomic data set is a set of data objects whose consistency can be maintained by a transaction, independently of how other atomic data sets are being updated by other transactions. We call this property of an atomic data set "consistency preserving". It is important to point out that consistency preserving implies neither that data objects in different atomic data sets are unrelated nor that data structures must be static. For example, we might have a document queue associated with each of the computers in a local network and several print queues associated with printing devices in the network. Each queue has its own consistency constraints in the form of "$0 \leq$ QueueSize $\leq$ MaxSize". There is a clear "producer-consumer" relation between document queues and printer queues. Nonetheless, each queue forms an atomic data set because the consistency of any queue can be satisfied independently of the others.

As an additional example showing the possible inter-relatedness of data objects in distinct atomic data sets,

let us consider a simplified model of a distributed directory system.[7] It consists of a set of local directories (LD) in the form of trees with consistency constraints that each entry in a LD must point to the correct location of files residing in the disk. We also have two global directories (GD) in the form linked lists of <FileName, NodeLocation> pairs with the consistency constraints that an entry in a GD must point to either the current location or a historical location of a file. In addition, each node maintains a partial global directory (PGD) that contains the location information of frequently used remote files and the forwarding addresses of migrated files. The data structure and consistency constraints of a PGD are the same as those of a GD. It is obvious that these directories are inter-dependent entities. However, each LD, PGD, and GD is an atomic data set, because we can maintain the consistency of anyone of them without blocking the activities of transactions on other atomic data sets. For example, we can readlock a LD, get the information we need, unlock it and then update a GD. That is, we can read the LD without blocking the activities on other atomic data sets, and we can update the GD and satisfy its consistency constraints, independently of how the LD or other atomic data sets are being updated by other transactions. Finally, linked lists, queues and trees are examples of atomic data sets that consist of dynamic data structures.

To illustrate compound transactions and their scheduling, let us consider the following example. Suppose that transaction Get-A-and-B needs one unit of some resource at node A and another unit at node B. This transaction requires both resource types, and if it cannot have both then it will not keep anyone of them. The resource heap at each node is modelled as an atomic data set with consistency constraint "$0 \leq$ HeapSize $\leq$ MaxSize", and the transaction is written in the form of a compound transaction, Get-A-and-B, which is illustrated in Table 5-1.

This example illustrates the following three characteristics of our approach. First, the database is partitioned into two consistency preserving atomic data sets: Heap A and Heap B. Second, the compound transaction Get-A-and-B consists of four correctness preserving elementary transactions, which co-operatively carry out the task of the compound transaction by passing information to each other via atomic variables *Obtain-A* and *Obtain-B*. As long as the post-conditions of the executed elementary transactions are individually satisfied, the post-conditions of the compound transaction Get-A-and-B are satisfied. In addition, each of these elementary transactions also preserves the consistency of Heap A and Heap B. Third, the locking protocol used by this transaction only ensures that each elementary transaction is executed serializ-

[7] For a detailed discussion of the distributed directory example, see Chapter 7 of [Sha 85a].

```
CompoundTransaction Get-A-and-B;
AtomicVariable Obtain-A, Obtain-B : Boolean;
BeginSerial
  BeginParallel
    ElementaryTransaction Get-A;
    BeginSerial
      Writelock Heap A;
      Take a unit from A if available
      and indicate the result in  atomic variable Obtain-A.
      Commit and unlock Heap A;
    EndSerial;

    ElementaryTransaction Get-B;
    BeginSerial
      WriteLock Heap B;
      Take a unit from B if available
      and indicate the result in atomic variable Obtain-B;
      Commit and unlock Heap B;
    EndSerial;
  EndParallel;

  BeginParallel;
    ElementaryTransaction Put-Back-A;
    BeginSerial
      If Obtain-A and not Obtain-B
      then BeginSerial
             WriteLock Heap A;
             Put-Back the unit of A;
             Commit and unlock Heap A;
           EndSerial;
    EndSerial;

    ElementaryTransaction Put-Back-B;
    BeginSerial;
      If Obtain-B and not Obtain-A
      then BeginSerial
             WriteLock Heap B;
             Put-Back the unit of B;
             Commit and unlock Heap B;
           EndSerial;
    EndSerial;
  EndParallel;
EndSerial.
```

Table 5-1: Compound Transaction Get-A-and-B

ably with respect to each atomic data set. Global serializability of concurrency control is not enforced. For example, suppose that we have another compound transaction Get-A-or-B, which is illustrated in Table 5-2. Transaction Get-A-or-B tries to get one unit of resource from node A and one unit of resource from node B. If it cannot get both, then it keeps whatever it has obtained.

```
CompoundTransaction Get-A-or-B;
BeginParallel
  ElementaryTransaction Get-A;
  BeginSerial
    Writelock Heap A;
    Take a unit from A if available;
    Commit and unlock Heap A;
  EndSerial;

  ElementaryTransaction Get-B;
  BeginSerial
    WriteLock Heap B;
    Take a unit from B if available;
    Commit and unlock Heap B;
  EndSerial;
EndParallel.
```

Table 5-2: Compound Transaction Get-A-or-B

Suppose that we execute transactions Get-A-and-B and Get-A-or-B with Heap A and Heap B with initial values of one. It is possible that transaction Get-A-and-B gets the only unit of A first, while transaction Get-A-or-B gets the only unit of B first. As a result, transaction Get-A-and-B will return a unit to A, while transaction Get-A-or-B will keep the unit obtained from B. The result of this execution is *not* equivalent to executing these two transactions serially. When these two transactions are executed serially, one of them will get both units.

Eventhough the transactions are not executed serializably, the consistency and correctness of concurrency control will be preserved. That is, the sizes of Heap A and Heap B will be positive and within their bounds, transaction Get-A-and-B will either get both units or nothing and transaction Get-A-or-B will either get both units, one of the two units or nothing. From an application point of view, the partition of the database and of transactions increases the system concurrency and reduces the probability of deadlocks.

The consistency and correctness of compound transactions like Get-A-and-B and Get-A-or-B is not an isolated example but rather the result of satisfying the generalized setwise serializable scheduling rule developed in this work. Readers who are interested in the use of this theory in the development of a decentralized operation system are recommanded to read Tokuda's, Clark's and Locke's work [Tokuda 85].

This chapter is organized as follows. In Section 5.2, we formally define the basic concepts of our model. In

Section 5.3, we formally define the concept of *atomic data sets* and develop our first modular scheduling rule called the *setwise serializable scheduling rule* and in Section 5.4 we develop a new transaction structure called *compound transactions* and their associated scheduling rules.

5.2 A Model of Modular Scheduling Rules

We begin our formal investigation by developing a model of modular scheduling rules. Intuitively, a scheduling rule specifies how the steps of a transaction can be interleaved with those of other transactions. In our model, a scheduling rule performs this specification by partitioning the steps of each transaction into equivalence classes called *atomic step segments.* Schedules satisfy the specification of a given scheduling rule by interleaving the atomic step segments of each transaction serializably with the atomic step segments of other transactions. For example, serializability theory is a particular scheduling rule which takes all the steps of a transaction as a single atomic step segment. Serializable schedules are the set of schedules that satisfy this rule, because the atomic step segments specified by serializability theory are interleaved serializably in serializable schedules.

This section is organized as follows. In Section 5.2.1, we define the concepts related to the notion of the database and in Section 5.2.2 we define the concepts of transactions and their schedules. In Section 5.2.3, we define the concepts of a scheduling rule and their properties.

5.2.1 Database

A database is simply a set of shared data objects, and a state of the database is a vector whose components are the values of these shared data objects. A consistent state is a state that satisfies a given set of consistency constraints. We now formalize the concepts related to the concept of database.

Definition 5.2.1-1: A data object, O, is a user defined smallest unit of data which is individually accessible and upon which synchronization can be performed (e.g. locking).

Definition 5.2.1-2: Associated with each data object O, we have a set Dom(O), the domain of O, consisting of all possible values taken by O.

Definition 5.2.2: Each data object is represented by triplets, <name, value, version number>. When a data object is created, its initial value is assigned to version zero of this data object, e.g. "A[0]: = 1". When the data object is updated, a new version of the object is created, and the transaction works on this new version.

In the following discussion, when we refer to the current value of a data object O, we would, for simplicity, write "O" instead of "O[v]". The version number representation will be used when different versions of the values of a data object are referred to.

Definition 5.2.3-1: The system database $D = \{O_1, O_2, ..., O_n\}$ is the collection of all the shared data objects in the system.

Definition 5.2.3-2: A state of database D is an n-tuple $Y \in \Omega = \prod_{j=1}^{n} Dom(O_j)$.

Definition 5.2.3-3: Associated with database D, there is a set of consistency constraints in the form of predicates on the states of database D. A consistent state of database D is an n-tuple, Y, satisfying this set of consistency constraints. This is indicated by "C(Y) = 1", where C is a Boolean function indicating whether this set of consistency constraints is satisfied by Y. For simplicity, we will also refer to this set of consistency constraints by C. The meaning of C is easily determined by the context. The set of all consistent states of D is denoted by U, where $U = \{Y \mid C(Y) = 1\}$.

5.2.2 Transactions and Their Schedules

Having defined the concept of the database, we now formalize the concept of transactions and schedules.

5.2.2.1 Transactions

In this section, we first define the syntax of single level transactions and the concepts of pre- and post-conditions of a transaction. We then enumerate our fundamental assumptions regarding transactions. In later sections, we will introduce generalizations to the single level transactions defined here.

Definition 5.2.4: A single level transaction T_i is a sequence of transaction steps $(t_{i,1}, t_{i,2}, ..., t_{i,m_i})$. A transaction step is modelled as the non-divisible execution of the following instructions [Kung 79]:

$$L_{t_{ij}} := O_{t_{ij}}$$

$$O_{t_{ij}} := f_{t_{ij}}(L_{t_{i,1}}, L_{t_{i,2}}, ..., L_{t_{ij}})$$

where the symbol "t_{ij}" represents step j of transaction T_i; the local variable $L_{t_{ij}}$ is used by step t_{ij} to store the value read. The symbol "$O_{t_{ij}}$" is the data object accessed (read or written) by step t_{ij}, and the symbol "$f_{t_{ij}}$" represents the computation performed by step t_{ij}.

In this model, every step reads and then writes a data object. A read step is interpreted as writing the value read back to the data object. That is, the function $f_{i,j}$ associated with a read step is the identity function. We now define the pre- and post-conditions of a transaction. We begin with defining the input steps and output steps in a transaction.

Definition 5.2.5: Let $T_i = \{t_{i,1}, ..., t_{i,m_i}\}$ be a transaction. Let data object O be the one accessed (read or written) by step $t_{i,j}$. Step $t_{i,j}$ is said to be an *input step* if it is the step in T_i that first accesses data object O. Step $t_{i,j}$ is said to be an *output step* if it is the step in T_i last accessing O. That is, for every data object O accessed by T_i there is an input step and an output step associated with O. Note that when there is only one step in T_i accessing O, then this step is both an input and an output step.

Since transactions operate on the shared database, they must be able to accept any consistent state as their input. That is, the values input to a transaction are assumed to satisfy the pre-conditions of the transactions as long as these values come from a consistent database state.

Definition 5.2.6: Let $O_{i,j}[v_b]$, $j = 1$ to k_i, be the set of values read by the input steps of T_i, where v_b denotes the version of a data object that is input to a transaction. Let the index set of $O_{i,j}[v_b]$, $j = 1$ to k_i, be I_m. The input values to T_i, $O_{i,j}[v_b]$, $j = 1$ to k_i, are said to satisfy the pre-condition of T_i, if and only if

$$\exists(X \in U)(\pi_{I_m}(X) = O_{i,j}[v_b], j = 1 \text{ to } k_i)$$

where π_{I_m} is the projection operator. That is, $\pi_{I_m}(X)$ is a tuple whose elements are those of a consistent X indexed by I_m. Having discussed the pre-conditions, we now turn to the subject of post-conditions.

Definition 5.2.7: Let $O_{i,j}[v_f]$, $j = 1$ to k_i, be the set of values written by the output steps of transaction T_i, where v_f denotes the version of a data object output by the transaction. The *post-condition* of transaction T_i is the specification of the output values of T_i as functions of the input values,

$$O_{i,j}[v_f] = g_{i,j}(O_{i,1}[v_b], ..., O_{i,k_i}[v_b]), j = 1 \text{ to } k_i.$$

Having defined concepts relating to the notion of a transaction, we now state our fundamental assumptions about a transaction:

Fundamental Assumptions:

- A1 *Termination*: A transaction is assumed to have a finite number of steps and is assumed to terminate.
- A2 *Transaction Correctness*: A transaction is assumed to produce results that satisfy its post-condition when executing alone and when the database is initially consistent.
- A3 *Transaction Consistency*: Given a consistent state of the database, a transaction is assumed to produce a state that satisfies the consistency constraints of the database when it is executing alone.

Definition 5.2.8: A transaction T_i is said to be consistent and correct if and only if T_i satisfies assumptions A1, A2 and A3.

In the following discussion, we restrict our attention to consistent and correct transactions.

5.2.2.2 Schedules

In the previous two sections, we have formalized the basic concepts related to database and transactions. We now develop our model one step further by considering the execution of a set of transactions. To this end, we introduce the concepts of transaction systems and schedules.

Definition 5.2.9: A transaction system T is a finite set of transactions $\{T_1, ..., T_n\}$ operating upon the shared database D.

Definition 5.2.10: A schedule z for transaction system T is a totally ordered set of all the steps in the transaction system $T = \{T_1, ..., T_n\}$ such that the ordering of steps of T_i, i = 1 to n, in the schedule is consistent with the ordering of steps in the transaction T_i, i = 1 to n.

$$[\forall(t)((t \in z) \Leftrightarrow (t \in \cup T))] \wedge [\forall(T_i \in T)\forall((t_{i,j}, t_{i,k} \in T_i) \wedge (t_{i,k} > t_{i,j}))((t_{i,j}, t_{i,k} \in z) \wedge (t_{i,k} > t_{i,j}))]$$

A schedule z for transaction system T is said to be consistent if and only if the execution of T according to z preserves the consistency of the database D. This concept is formalized as follows.

Definition 5.2.11: Let X be the initial state of D and Y be the state at the end of executing T according to z. Schedule z is said to be consistent if and only if

$z: X \in U \rightarrow Y \in U$

A schedule z for a transaction system T is said to be correct if and only if the execution of transactions in T according to z produces computations that satisfy the post-condition of each of the transactions in the system.

Definition 5.2.12: Under schedule z, let the values input to and output from transaction $T_i \in T$ be $O_{i,1}[v_b]$, ... $O_{i,k_i}[v_b]$ and $O_{i,1}[v_f]$, ... $O_{i,k_i}[v_f]$ respectively. Schedule z is said to be correct if and only if

$$\forall(T_i \in T)(O_{i,j}[v_f] = f_{i,j}(O_{i,1}[v_b], ..., O_{i,k_i}[v_b]), j = 1 \text{ to } k_i.)$$

Having defined the basic properties of a schedule, we now consider the relations between schedules by introducing the concept of equivalent schedules. Conceptually, two schedules z and z^* for transaction system T are equivalent if for any given initial state of D the executions of T according to z and z^* yield the same sequences of values for each data object in the database and the same sequences of values for each of the local variables (states) of each transaction in T. This is formally defined by the partial ordering of steps induced by z and z^* on each of the data objects in D.

Definition 5.2.13: Let $O_{t_{i,j}} \equiv O_{t_{k,m}}$ denote step $t_{i,j}$ and step $t_{k,m}$ read or write the same data object. A schedule z for transaction system T is said to be *equivalent* to another schedule z^* for T, if for every pair of steps $t_{i,j}$ and $t_{k,m}$ in z and z^*,

$$\forall((t_{i,j}, t_{k,m} \in \cup T) \wedge (O_{t_{i,j}} \equiv O_{t_{k,m}})) [((t_{i,j}, t_{k,m} \in z) \wedge (t_{i,j} > t_{k,m})) \Leftrightarrow ((t_{i,j}, t_{k,m} \in z^*) \wedge (t_{i,j} > t_{k,m}))]$$

That is, for each of the data objects in the database, O, the orderings of transaction steps induced by z and by z^*, $t_{k,m}(... (t_{i,j}(O))$, are identical.

We now prove an important theorem which states that if z and z^* for T are equivalent, and if z is consistent and correct, then z^* is also consistent and correct. We begin our proof with the following lemma.

Lemma 5.2.1: Let z and z^* be two schedules for transaction system T. Let t be a step of transaction $T_i \in T$. Let the values of the data object accessed by t in z and z^* be O_t and O_t^* respectively. Let the values of the local variable associated with t in z and z^* be L_t and L_t^* respectively. Given the identical initial state X to both z and z^*, we have

$$\forall(T_i \in T)\forall(t \in T_i)((O_t = O_t^*) \wedge (L_t = L_t^*))$$

That is, the values input to and output from any transaction step under z are equal to those under z^*.

Proof: Recall that the syntax of a transaction step is as follows.

$$L_{t_{i,l}} := O_{t_{i,l}}$$

$$O_{t_{i,l}} := f_{t_{i,l}}(L_{t_{i,1}}, ..., L_{t_{i,l}})$$

It follows that a transaction step will output identical values under z and z^* if for any transaction step $t \in \cup T$ the values input to t under z and under z^* are equal. Thus, we need to show only that each transaction step $t \in \cup T$ inputs the same values under both schedules z and z^*.

Now consider the first step in schedule z denoted as $t_{z,1}$. Step $t_{z,1}$ must input the initial value of some data object in D denoted as $O_{t_{z,1}}$. By the definition of equivalent schedules and with the same initial state of D, step $t_{z,1}$ must input the same initial value of data object $O_{t_{z,1}}$. Hence, $t_{z,1}$ outputs the same value under both z and z^*. Now consider the second step $t_{z,2}$ in z. The value input to the local variable of $t_{z,2}$ under z is either the initial value of a data object or the value output by step $t_{z,1}$. By the definition of equivalent schedules and by the fact that $t_{z,1}$ outputs the same value under both z and z^*, step $t_{z,2}$ will input the same value under both z and z^*. Suppose that step $t_{z,h}$ inputs the same value under schedules z and z^*. Following the argument above, step $t_{z,h+1}$ inputs the same value under z and z^*. The lemma follows by induction. □

Theorem 5.2.1: Let schedule z be a consistent and correct schedule for transaction system T. If schedule $z^* \equiv z$, then z^* is also consistent and correct.

Proof: To show that z^* is correct, we need to show that for any $T_i \in T$, the execution of T_i under z^* produces correct results. Let the values input to T_i be $O_{i,1}[v_b]$, ..., $O_{i,k_i}[v_b]$. Let the values output by T_i be $O_{i,1}[v_f]$, ..., $O_{i,k_i}[v_f]$. The post-condition of T_i is the specification of the output values of T_i as some functions of the input values: $O_{i,j}[v_f] = f_{i,j}(O_{i,1}[v_b], ..., O_{i,k_i}[v_b])$, $j = 1$ to k_i. It follows from Lemma 5.2.1 that all the input values to and the output values from T_i under z and z^* are identical. Therefore, the post-condition of T_i must be satisfied under both z and z^*.

To show that z^* is consistent, let the initial state of D be $X \in U$ and the final states of D resulting from executing T according to z and z^* be Y and Y^* respectively. It follows from the definition of equivalent schedules and Lemma 5.2.1 that $Y = Y^*$. Thus, Y^* is a consistent state. □

As an example of equivalent schedules, a serializable schedule is defined as a schedule z for which there exists a serial schedule z^* such that $z \equiv z^*$. The consistency and correctness of serial schedules directly follow from the assumption that each transaction is consistent and correct when executing alone. By Theorem 5.2.1, serializable schedules are also consistent and correct.

5.2.3 Modular Scheduling Rules

Having defined the concepts of transactions and schedules, we now formally define the concepts related to modular scheduling rules.

5.2.3.1 Definition of Scheduling Rules

We now formalize the concept of a scheduling rule and its relation to schedules.

Let $\mathbb{T}_m$ denote the set of all the possible consistent and correct transactions with m steps. Let $\mathbb{T}$ denote the set of all the possible consistent and correct transactions, that is, $\mathbb{T} = \cup_{m=1}^{\infty} \mathbb{T}_m$. Let P_m denote a partition into atomic step segments of a m-step consistent and correct transaction. Let $\mathcal{P}_m$ denote the set of all the possible partitions of a consistent and correct m-step transaction. Let $\mathcal{P}$ be the set of all the possible partitions, that is, $\mathcal{P} = \cup_{m=1}^{\infty} \mathcal{P}_m$.

Definition 5.2.14-1: A scheduling rule for a transaction system with n transactions, R_n, is a function which takes the transaction system of size n and partitions each of the n transactions,

$$R_n: \prod_{i=1}^{n} \mathbb{T} \rightarrow \prod_{i=1}^{n} \mathcal{P}$$

Definition 5.2.14-2: A scheduling rule R is a function which takes a transaction system of any size and partitions each of the transactions in the system.

$$R: \cup_{n=1}^{\infty} (\prod_{i=1}^{n} \mathbb{T}) \rightarrow \cup_{n=1}^{\infty} (\prod_{i=1}^{n} \mathcal{P})$$

such that the restriction of R to $\prod_{i=1}^{n} \mathbb{T}$ is R_n. That is, $R|\prod_{i=1}^{n} \mathbb{T} = R_n$, n = 1 to ∞

Given a scheduling rule R, we must identify the set of schedules that satisfy R. A schedule z satisfies R if atomic step segments of one transaction are interleaved serializably with those of others in z. This is formalized as follows.

Definition 5.2.15: Let $T = \{T_1, ..., T_n\}$ be a consistent and correct transaction system, that is, $T \subseteq \mathcal{T}$. Let T_i be a transaction in T. Let $\Xi_R(T_i)$ denote the partition of the steps of T_i by R. Let Γ be the set of all the atomic step segments of T specified by R, that is, $\Gamma = \cup_{T_i \in T} \Xi_R(T_i)$. Let D be the database and $Z(T)$ be the set of all the possible schedules for T. Finally, let σ_i be an atomic step segment of T_i, that is, $\sigma_i \in \Xi_R(T_i)$. A schedule $z \in Z(T)$ is said to be *atomic step segment serial* with respect to R if atomic step segments specified by R and belonging to different transactions do not overlap in z. That is,

$$\forall(T_i, T_j \in T, i \neq j)\, \forall(\sigma_i \in \Xi_R(T_i))\, \forall(\sigma_j \in \Xi_R(T_j))\, \forall(t \in \sigma_i)\, ((t < t_j^1) \vee (t > t_j^m))$$

where t_j^1 and t_j^m are the first and last steps in σ_j respectively.

Definition 5.2.16: A schedule $z \in Z(T)$ is said to satisfy scheduling rule R, if atomic step segments specified by R and belonging to different transactions are interleaved serializably in z. That is, z satisfies R if and only

$$\exists(z^* \in Z(T))\, ((z \equiv z^*) \wedge (z^* \text{ is atomic step segment serial}))$$

The set of all the schedules in Z(T) that satisfies R is denoted by $Z_R(T)$. That is,

$$Z_R(T) = \{z \in Z(T) \mid z \text{ satisfies } R\}$$

5.2.3.2 Consistency and Correctness

In the previous section, we have defined the concept of a scheduling rule and its relation to schedules. We now define the key properties of a scheduling rule: consistency and correctness.

Definition 5.2.17: A scheduling rule R is said to be consistent and correct if and only if all the schedules that satisfy R are consistent and correct,

$$\forall(T \subseteq \mathcal{T}) \forall(z \in Z_R(T))\, (z \text{ is consistent and correct})$$

where $\mathbf{T}$ is the set of all the consistent and correct transactions. In the following, we limit our discussions to consistent and correct scheduling rules.

5.2.3.3 Modularity, Optimality and Completeness

We consider a scheduling rule to be *modular*, if it partitions each transaction in a way that is independent of all the other transactions in the system. We consider a scheduling rule to be optimal under some condition, if this rule provides the highest degree of concurrency under this condition. We consider a set of consistent and correct modular scheduling rules to be complete if for any consistent and correct modular scheduling rule **R**, we can always find a scheduling rule $\mathbf{R}^*$ in the set such that $\mathbf{R}^*$ provides at least much concurrency as **R**.

Definition 5.2.18: A scheduling rule **R** is said to be *modular* if and only if **R** partitions each transaction independently. That is,

$$\mathbf{R}(T_1, ..., T_n) = (R_1^1(T_1), ..., R_1^n(T_n)),\ n = 1 \text{ to } \infty,$$

The scheduling rule for individual transactions, $\mathbf{R}_1^i$, i = 1 to n, will be referred to as the *kernels* of the modular scheduling rule **R**. Note that the kernels of a modular scheduling rule need not be identical for different transactions. In other words, a modular scheduling rule can consist of a family of kernels. This allows each of these kernels to take advantage of the semantic information of the given transaction.

Having defined the concept of modularity, we come to address issues related to the degree of concurrency provided by scheduling rules. We address these issues using the concepts of optimality and completeness. Before formally defining these two concepts, we first comment on the implications of using a richer set of primitive steps in addition to the "read" and "write" steps used in this work. Korth [Korth 83] has shown that for serializable schedules the concurrency can be improved if the set of primitive steps is expanded to include other commutative ones. The idea is that if a set of steps is commutative, then there is no need to control their relative order. In this chapter, we limit our discussion to transactions using only the primitive steps: "read" and "write". The use of commutative steps to improve the concurrency of (generalized) setwise serializable schedules can be done in a manner similar to that done by Korth for serializable schedules. We begin our investigation by first defining a way to compare the degree of concurrency offered by different scheduling rules.

Definition 5.2.19: Scheduling rule R^1 is said to be *at least as concurrent as* R^2, denoted by $R^1 \geq R^2$, if and only if,

$$\forall(T \subseteq T)(Z_{R^2}(T) \subseteq Z_{R^1}(T))$$

That is, the concurrency of schedules is partially ordered by set containment. We now define the concept of an optimal modular scheduling rule.

Definition 5.2.20: Let Λ_M be the set of all the consistent and correct modular scheduling rules. A modular scheduling rule $R^* \in \Lambda_M$ is said to be optimal if R^* is at least as concurrent as any rule in Λ_M. That is,

$$\forall(R \in \Lambda_M)(R^* \geq R)$$

We now introduce the concept of *completeness*. A family of modular scheduling rules is said to form a complete class within the set of modular scheduling rules if and only if given any modular scheduling rule R we can always find a rule R^* in this family of rules such that R^* is at least as concurrent as R.

Definition 5.2.21: Let Λ_M be the set of all the consistent and correct modular scheduling rules. A set of consistent and correct modular scheduling rules $\mathfrak{R}$ is said to form a *complete class* within Λ_M, if and only if

$$\forall(R \in \Lambda_M)[\exists(R^* \in \mathfrak{R})(R^* \geq R)]$$

5.3 The Setwise Serializable Scheduling Rule

Having developed a formal model of modular scheduling rules, we now introduce an important modular scheduling rule called the setwise serializable scheduling rule. In essence, this rule states that if the database is partitioned into consistency preserving *atomic data sets*, then we can replace the global serializability of concurrency control by setwise serializability. Intuitively, an atomic data set is simply a set of data objects with a set of associated consistency constraints that can be satisfied independently of other atomic data sets. For example, in a distributed computer system a job queue at a computer waiting for execution is an atomic data set, the local directory to a computer's files is an atomic data set, and each user's mailbox is also an atomic data set. This is because the consistency of each of these entities is defined by a set of consistency constraints that can be satisfied independently of others.

Proof: First, if any atomic step segment σ in T_i specified by **R** does not preserve the consistency of the database, then another transaction T_j executing after σ would input an inconsistent state. Since **R** is modular, we can define the semantics of T_j as one that outputs incorrect results when its input is inconsistent. Thus **R** is incorrect. Second, if the conjunction of the post-conditions of all the atomic step segments in T_i specified by **R** is not equivalent to the post-condition associated with T_i, then **R** is incorrect by definition. Since any schedule z for any transaction system $T \subseteq \mathbf{T}$ satisfying **R** guarantees the serializability in the execution of the transaction atomic step segments specified by **R**, it follows from conditions 1 and 2 that z is consistent and correct. □

Theorem 5.4.3: Generalized setwise serializable scheduling rules form a *complete class* within the set of modular scheduling rules.

Proof: Let **R** be a consistent and correct modular scheduling rule. Suppose that T_i is a consistent and correct transaction and T_i is partitioned into atomic step segments σ_1, ..., σ_k by **R**. First, by Lemma 5.4.3 σ_i, i = 1 to k, must preserve the consistency of the database when executing alone. Second, by Lemma 5.4.3 the conjunction of the post-conditions of σ_1, ..., σ_k must be equivalent to the post-conditions associated with T_i. Note that each σ_i, i = 1 to k, satisfies the definition of an elementary transaction. We now define a generalized setwise serializable scheduling rule $\mathbf{R}^*$ which partitions T_i as follows. First, $\mathbf{R}^*$ labels σ_1, ..., σ_k as elementary transactions. Next, $\mathbf{R}^*$ partitions these elementary transactions into transaction ADS segments. Hence, $\mathbf{R}^*$ is at least as concurrent as **R**. □

5.5 Conclusion

In this chapter, we have developed a formal theory of modular scheduling rules. This theory coherently addresses the notions of consistency, correctness, modularity and completeness in concurrency control. Furthermore, this theory provides us with a compete set of provably consistent, correct and modular concurrency control rules. From an application point of view, this means that if each programmer can ensure the consistency and correctness of his transaction, and he follows the rules given in the chapter, then the consistency and correctness of system concurrency control is also ensured. In addition, our approach provides at least as much concurrency as any other consistent and correct modular concurrency control method.

Finally, we want to cite a few results from [Sha 85a] that we omitted in this chapter. First, the setwise

Thus far, we have shown that the generalized setwise serializable scheduling rule R is consistent, correct and modular. We now prove that generalized setwise scheduling rules (i.e. the kernels of R) form a complete class within the set of all the consistent and correct modular scheduling rules. Before proceeding with the proof of completeness, we need to introduce the concept of the post-conditions associated with an atomic step segment. To illustrate the need, let the transaction $T_i = \{t_{i,1}: \Lambda := \Lambda - 1; t_{i,2}: \Lambda := \Lambda + 1\}$. If the two steps of T_i are treated as a single atomic step segment, then $t_{i,1}$ is an input step and $t_{i,2}$ is an output step. The partition of a transaction could create input and output steps in addition to those defined in an executing alone environment. For example, if each of these two steps is an atomic step segment, then $t_{i,1}$ ($t_{i,2}$) is both an input step and an output step.

Definition 5.4.7: Let $\sigma = \{t_{i,1}, ..., t_{i,k}\}$ be an atomic step segment. Let the data object accessed by step $t_{i,j} \in \sigma$ be O. Step $t_{i,j}$ is an input step if it is the step in σ first accessing O. Step $t_{i,j}$ is an output step if it is the step in σ last accessing O.

Definition 5.4.8: Let $O_j[v_b]$, j = 1 to k, be the values input to the input steps of σ and $O_j[v_f]$, j = 1 to k, be the values output by the output steps of σ. The post-condition of σ is a specification of the output values as functions of input values and the values of the local variables associated with the steps of preceding atomic step segments of the same transaction. That is,

$$O_j[v_f] = f_j(L_j, O_1[v_b], ..., O_k[v_b]), j = 1 \text{ to } k;$$

where L_j is the set of local variables associated with the steps of the atomic step segments preceding the output step for O_j.

We now prove that generalized setwise serializable scheduling rules form a complete class within the set of all the consistent and correct modular scheduling rules.

Lemma 5.4.3: Let R be a modular scheduling rule. R is consistent and correct if and only if

1. for each of the transactions T_i in T, the conjunction of the post-conditions of all the atomic step segments in T_i is equivalent to the post-condition associated with T_i.

2. let z be a schedule for a transaction system $T \subseteq \mathbb{T}$ and z satisfies R. In the execution of T according to z, any atomic step segment in $T_i \in T$ specified by R preserves the consistency of the database.

5.4.2 Generalized Setwise Serializable Scheduling Rules

Having defined the syntax of a compound transaction, we now define the concept of generalized setwise serializable scheduling rules.

Definition 5.4.5 Let $T = \{T_1, ..., T_n\}$ be a transaction system. The generalized setwise serializable scheduling rule R is a modular scheduling rule with the following properties.

1. $R(T_1, ..., T_n) = (R_1^1(T_1), ..., R_1^n(T_n))$
2. The kernel of R, R_1^i, i = 1 to n, is a composite function which first maps T_i into a compound transaction T_i^c, and then partitions the steps of each of the elementary transactions of T_i^c into transaction ADS segments.

For simplicity, we refer to the kernels of the generalized setwise serializable scheduling rule as generalized setwise serializable scheduling rules.

Definition 5.4.6: A schedule z for a transaction system T is said to be generalized setwise serializable if z satisfies the generalized setwise serializable scheduling rule R. That is, the transaction ADS segments specified by R in one elementary transaction are interleaved serializably with those of others in z.

Theorem 5.4.1: Generalized setwise serializable schedules are consistent and correct.

Proof: Since a generalized setwise serializable schedule is setwise serializable with respect to all the elementary transactions in the system, it follows from Assumption 5.4.1 and Corollary 5.3.3-1 that each elementary transaction terminates, preserves the consistency of the database and produces results that satisfy its post-conditions. Hence, the consistency of the database is preserved and the post-condition of each of the compound transactions is also satisfied. Therefore, generalized setwise serializable schedules are consistent and correct. □

Corollary 5.4.1: Generalized setwise serializable scheduling rules are consistent and correct.

Theorem 5.4.2: Generalized setwise serializable scheduling rules are modular.

Proof: It directly follows from Definitions 5.2.18 and 4.5. □

$$O_{t_{i,j}} = f_{t_{i,j}}(L_p, l_{t_{i,1}}, ..., l_{t_{i,j}})$$

where L_p is the set of local variables associated with all the preceding elementary transactions in the same compound transaction.

We now define the concepts of input and output steps of an elementary transaction. Next, we define the concept of the post-conditions of an elementary transaction.

Definition 5.4.3: Step $t_{i,j} \in T_i^e$ is an input step if it is the step in T_i^e first accessing a data object O. Step $t_{i,j}$ is an output step if it is the step in T_i^e last accessing O.

Definition 5.4.4: Let $O_j[v_b]$, j = 1 to k, be the values input to the input steps of T_i^e and $O_j[v_f]$, j = 1 to k, be the values output by the output steps of T_i^e. The post-condition of T_i^e is a specification of the output values as functions of input values and the values of local variables associated with the steps in the preceding elementary transactions.

$$O_j[v_f] = f_j(L_p, O_1[v_b], ..., O_k[v_b]), j = 1 \text{ to } k.$$

where L_p is the set of local variables associated with the steps in the preceding elementary transactions of the same compound transaction.

Having defined the syntax of compound transactions, we now state our assumptions about them.

Assumption 5.4.1: When an elementary transaction of a compound transaction is executed serially and in an order that is consistent with the partial order defined by the compound transaction, it satisfies our three fundamental assumptions about a transaction, that is, it terminates (A1), preserves the consistency of the database (A2) and satisfies its own post-conditions (A3). Furthermore, the post-condition of a compound transaction is equivalent to the conjunction of all the post-conditions of its elementary transactions.

5.4 Compound Transactions

A compound transaction consists of a partially ordered set of elementary transactions. When an elementary transaction is executed serializably and in an order consistent with the partial ordering of the elementary transactions in the compound transaction, it has the following two properties. First, given a consistent state of the database and the results passed from preceding elementary transactions, an elementary transaction produces another consistent state of the database and satisfies its own post-conditions. Second, the conjunction of the post-conditions of the constituent elementary transactions is equivalent to the post-conditions of the compound transactions. When the database is partitioned into atomic data sets and transactions are partitioned in the form of compound transactions, the consistency and correctness of concurrency control will be ensured as long as the elementary transactions of the compound transactions are run setwise serializably. This result is formally expressed as the generalized setwise serializable scheduling rules. We also prove that our approach of partitioning the database and transactions provides as least as much concurrency as any other modular concurrency control approach. This optimality result is formally expressed as the completeness of generalized setwise serializable scheduling rules.

5.4.1 Syntax

We begin our formal investigation of compound transactions by first defining their syntax. A compound transaction consists of a partially ordered set of elementary transactions. Each elementary transaction can have the structure of a nested transaction. These elementary transactions collectively carry out the task of the compound transaction by passing information via local variables. For simplicity, we assume that elementary transactions are single level transactions in this chapter.[8]

Definition 5.4.1: A compound transaction is a partially ordered set of elementary transactions.

Definition 5.4.2: Let $T_i^e = \{t_{i,1}, ..., t_{i,k}\}$ be an elementary transaction. Let the data object accessed by step $t_{i,j} \in T_i^e$ be $O_{t_{i,j}}$. Let the local variable associated with step $t_{i,j}$ be $L_{t_{i,j}}$. Step $t_{i,j}$ is modelled by the indivisible operation of the following two instructions.

$$L_{t_{i,j}} := O_{t_{i,j}}$$

[8] For nested transactions, see Sha's thesis [Sha 85a].

Proof: Let $Q = \{\mathcal{A}_1, ..., \mathcal{A}_k\}$ be a CP partition of D. Let the initial states of each of the ADS's be $Z_{A_j}[0]$, j = 1 to k. These initial states are assumed to be consistent.

Since a schedule is a totally ordered set of steps from all the transactions, each of which terminates, there must exist a transaction ADS segment $\Psi(i, \mathcal{A}_j)$ which first finishes its computation. Let the associated ADS state be $Z_{A_j}[1]$. Since there are no interleavings among transaction ADS segments accessing the same ADS in a setwise serial schedule, $Z_{A_j}[1]$ must be output by a transaction which has used only the initial states that were assumed to be consistent. By Lemma 5.3.3-3, $Z_{A_j}[1]$ is consistent, and the values of data objects in $\mathcal{A}_j$ output by $\Psi(i, \mathcal{A}_j)$ are correct. Consider now the output of the second transaction ADS segment produced by the schedule. Since it can use only $Z_{A_j}[1]$ or $Z_{A_m}[0]$, m = 1 to k and $m \neq j$, at the end of this second transaction ADS segment, the accessed atomic data set is in a consistent state and the output values are correct by Lemma 5.3.3-3. Now assume that the first n transaction ADS segments produce consistent and correct results. The $n+1^{st}$ must also by the same argument. By induction, the ADS state produced by each of the transaction ADS segments is consistent, and the values of the data objects output in each ADS at the end of the transaction ADS segment satisfy the post-condition. It follows that a setwise serial schedule is consistent and correct. □

Corollary 5.3.3-1: Setwise serializable schedules are consistent and correct.

Proof: It directly follows from Definition 5.3.6 and Theorem 5.2.1. □

Corollary 5.3.3-2: The setwise serializable scheduling rule is consistent and correct.

Proof: It directly follows from Corollary 5.3.3-1. □

Throughout this section, the choice of atomic data sets has been arbitrary, because Theorem 5.3.3 applies to any CP partition whether maximal or not. If the CP partition consists of a single ADS, then setwise serializable schedules reduce to serializable schedules.

Lemma 5.3.3-3: In a setwise serial schedule, if at the beginning of a transaction ADS segment, $\Psi(i, \mathcal{A}_{i,j})$, = 1 to k_i, ADS $\mathcal{A}_{i,j}$ is initially consistent, then $\mathcal{A}_{i,j}$ is consistent at the end of $\Psi(i, \mathcal{A}_{i,j})$, and the values of each of the data objects in $\mathcal{A}_{i,j}$ output by T_i are correct.

Proof: Let the atomic data sets accessed by T_i be $\mathcal{A}_{i,j}$, j = 1 to k_i. Now let T_i execute alone in a serial schedule z^* with the initial states of $\mathcal{A}_{i,j}$, j = 1 to k_i, being identical to the initial states of $\mathcal{A}_{i,j}$, j = 1 to k_i, in the setwise serial schedule z.

Let the values of the local variables and data objects in the serial schedule z^* be $L^*_{t_{i,l}}$ and $O^*_{t_{i,l}}$; and those in setwise serial schedule z be $L_{t_{i,l}}$ and $O_{t_{i,l}}$. We now prove that the executions of T_i under z and z^* are equivalent. Recall that the syntax of a transaction step is given by

$$L_{t_{i,l}} := O_{t_{i,l}}$$

$$O_{t_{i,l}} := f_{t_{i,l}}(L_{t_{i,1}}, ..., L_{t_{i,l}})$$

Since the initial states of ADS $\mathcal{A}_{i,j}$, j = 1 to k_i, are equal in both schedules, the initial values of all the data objects in $\mathcal{A}_{i,j}$, j = 1 to k_i, are equal. Therefore, the first steps in both schedules input the same value. That is, $L_{t_{i,1}} = L^*_{t_{i,1}}$. In addition, $O_{t_{i,1}} = f_{t_{i,1}}(L_{t_{i,1}}) = f_{t_{i,1}}(L^*_{t_{i,1}}) = O^*_{t_{i,1}}$. Next, $L_{t_{i,2}} = L^*_{t_{i,2}}$, because step two either reads the initial value of a data object or the value of the data object output by step 1. Similarly, $O_{t_{i,2}} = O^*_{t_{i,2}}$.

Now suppose that these local variable and data object value pairs are equal from steps 1 to r. That is, $L_{t_{i,h}} = L^*_{t_{i,h}}$ and $O_{t_{i,h}} = O^*_{t_{i,h}}$, h = 1 to r. We show that $L_{t_{i,r+1}} = L^*_{t_{i,r+1}}$. This follows because step r+1 either reads the initial value of a data object or a data object which has been output by some steps between 1 to r. It follows that $O_{t_{i,r+1}} = O^*_{t_{i,r+1}}$. Therefore, the final values of accessed data objects in both schedules are equal at the end of each transaction ADS segment. In addition, data objects in $\mathcal{A}_{i,j}$, j = 1 to k_i, not accessed by $T_{i,j}$ remain unchanged and therefore equal at the end of each transaction ADS segment for both schedules. It follows from Lemma 5.3.3-2 that at the end of $\Psi(i, \mathcal{A}_{i,j})$, the $\mathcal{A}_{i,j}$, j = 1 to k_i, are consistent, and the value of each of the data objects in $\mathcal{A}_{i,j}$, j = 1 to k_i, output by T_i is correct. □

Theorem 5.3.3: A setwise serial schedule is consistent and correct.

$L_{t_{i,l}} := O_{t_{i,l}}$

$O_{t_{i,l}} := f_{t_{i,l}}(L_{t_{i,1}}, ..., L_{t_{i,l}})$

Since the initial states of all accessed ADS are equal with either X or Y as the initial state, it follows that the initial values of the accessed data objects are equal. Hence, at the first step of T_i, $L_{t_{i,1}} = L^*_{t_{i,1}}$. In addition, $O_{t_{i,1}} = f_{t_{i,1}}(L_{t_{i,1}}) = f_{t_{i,1}}(L^*_{t_{i,1}}) = O^*_{t_{i,1}}$. Next, $L_{t_{i,2}} = L^*_{t_{i,2}}$, because the second step either reads the initial value of a data object or the value of the data object output by step 1. Similarly, $O_{t_{i,2}} = O^*_{t_{i,2}}$.

Now suppose that these local variable and data object value pairs are equal from steps 1 to r. That is, $L_{t_{i,h}} = L^*_{t_{i,h}}$ and $O_{t_{i,h}} = O^*_{t_{i,h}}$, h = 1 to r. We show that $L_{t_{i,r+1}} = L^*_{t_{i,r+1}}$. This follows because step r+1 either reads the initial value of a data object or a data object which has been output by some step between 1 to r. It follows that $O_{t_{i,r+1}} = O^*_{t_{i,r+1}}$. By induction, the final values of accessed data objects with either X or Y as initial state are equal. Since the values of data objects in $\mathcal{A}_{i,j}$, j = 1 to k_i, not accessed by T_i remain unchanged, they must be equal at the end of the transaction with either X or Y as the initial state. That is, $\pi_{S_{i,j}}(W_x) = \pi_{S_{i,j}}(W_y)$, j = 1 to k_i, where W_x and W_y are the states of D at the end of executing T_i with X and Y as initial states respectively. Since the execution using X as the initial state is assumed to preserve the consistency of each of the accessed atomic data sets, the execution using Y as the initial state must also preserve the consistency of each of the accessed atomic data sets. Since the execution with X as the initial state produces correct results, the execution using Y as the initial state must also produce correct results. □

Lemma 5.3.3-2: Let the atomic data sets accessed by transaction T_i be $\mathcal{A}_{i,j}$, j = 1 to k_i. If $\mathcal{A}_{i,j}$, j = 1 to k_i, are initially consistent and if T_i executes alone, then at the end of transaction ADS segment $\Psi(i, \mathcal{A}_{i,j})$ the consistency of $\mathcal{A}_{i,j}$ is preserved. Furthermore, the values of data objects in $\mathcal{A}_{i,j}$ output by $\Psi(i, \mathcal{A}_{i,j})$ are correct at the end of $\Psi(i, \mathcal{A}_{i,j})$.

Proof: At the end of the transaction ADS segment $\Psi(i, \mathcal{A}_{i,j})$, j = 1 to k_i, the data objects in $\mathcal{A}_{i,j}$, j = 1 to k_i are neither read or written again. It follows that the values of the data objects in $\mathcal{A}_{i,j}$, j = 1 to k_i, are the same as at the end of the transaction. By Lemma 5.3.3-1, at the end of the transaction, the consistency of each of the atomic data sets is preserved, and the values of the data objects output by T_i are correct, it follows that at the end of each of the transaction ADS segments the state of the accessed atomic data set is consistent and the values of the data objects output by the segment are correct. □

5.3.2.2 Consistency and Correctness

We now prove that setwise serializable schedules are consistent and correct. The proof is organized into three lemmas. Let T_i be a consistent and correct transaction. In Lemma 5.3.3-1, we prove that T_i preserves the consistency of each of the accessed atomic data sets and produces correct results when executing alone. In Lemma 5.3.3-2, we further prove that at the end of executing a transaction ADS segment $\Psi(i, \mathcal{A})$ of T_i, the consistency of $\mathcal{A}$ has been already preserved. In addition, the output values of data objects in $\mathcal{A}$ are correct at the end of $\Psi(i, \mathcal{A})$. We need not wait for the end of T_i to know these results. In Lemma 5.3.3-3, we relax the executing alone condition. We show that the results of Lemma 5.3.3-2 are still valid for any ADS $\mathcal{A}$, as long as $\mathcal{A}$ is consistent at the beginning of transaction segment $\Psi(i, \mathcal{A})$.

Definition 5.3.7: An ADS $\mathcal{A}_j$ is said to be accessed by a transaction, if this transaction reads or writes one or more data objects in $\mathcal{A}_j$.

Lemma 5.3.3-1: Let $Q = \{\mathcal{A}_1, ..., \mathcal{A}_k\}$ be a given CP partition of D. Let T_i be a consistent and correct transaction. If T_i executes alone and if the states of the atomic data sets accessed by T_i are initially consistent, then at the end of T_i the state of each of the accessed atomic data sets is consistent, and the values output by T_i satisfy the post-condition of T_i.

Proof: let $\mathcal{A}_{i,j}$, j = 1 to k_i, be the atomic data sets accessed by transaction T_i. Let $Y \in \Omega$ be a state of D such that $C_{i,j}(\pi_{S_{i,j}}(Y)) = 1$, j = 1 to k_i; where $C_{i,j}$ represents the ADS consistency constraints of $\mathcal{A}_{i,j}$, and $S_{i,j}$ represents the index set of $\mathcal{A}_{i,j}$. Now let X be a consistent state of the database such that $\pi_{S_{i,j}}(X) = \pi_{S_{i,j}}(Y)$, j = 1 to k_i. Next, we let T_i execute alone with the database initially in state X. We now prove that with either X or Y as initial state, the executions of T_i are equivalent.

By assumptions A1 to A3 in Section 2.2.1, with X as initial state, T_i produces correct results and preserves the consistency of the database. It follows that T_i preserves the consistency of all the atomic data sets. To complete the proof, we must show that the values of the local variables and the values of the global variables of T_i are identical in both schedules.

Let the values of the local variable and the data object in the execution with initial state X be $L^*_{t_{i,l}}$ and $O^*_{t_{i,l}}$. Let the values of the local variable and the data object with initial state Y be $L_{t_{i,l}}$ and $O_{t_{i,l}}$. We must prove that $L_{t_{i,l}} = L^*_{t_{i,l}}$ and $O_{t_{i,l}} = O^*_{t_{i,l}}$ at each step of the transaction. Recall that the syntax of a transaction step is as follows,

5.3.2 The Setwise Serializable Scheduling Rule

In this section, we first define this rule and then prove that this rule is consistent and correct.

5.3.2.1 Definitions

In this section, we first define the concept of a transaction ADS segment. We then define the scheduling rule called the *setwise serializable* scheduling rule. Finally, we define the set of schedules that satisfy this rule.

Definition 5.3.3: A transaction ADS segment is the sequence of steps in a transaction that read or write data objects in the same ADS. Let $\Psi(i, \mathcal{A})$ denote the transaction ADS segment of transaction T_i accessing ADS $\mathcal{A}$. Let $t_{i,j} > t_{k,m}$ denote that step $t_{i,j}$ is executed after step $t_{k,m}$. We have

1. $\Psi(i, \mathcal{A}) = \{t \mid (t \in T_i) \wedge (t \text{ reads or writes a data object in ADS } \mathcal{A})\}$

2. $\forall((t_{i,j}, t_{i,k} \in \Psi(i, \mathcal{A})) \wedge (t_{i,j} > t_{i,k}))((t_{i,j}, t_{i,k} \in T_i) \wedge (t_{i,j} > t_{i,k}))$

Definition 5.3.4: The scheduling rule that partitions each of the transactions in a transaction system T into transaction ADS segments is called the *setwise serializable* scheduling rule.

A setwise serial schedule z for a transaction system T is a schedule in which transaction ADS segments accessing the same ADS do not overlap. This concept is formalized as follows.

Definition 5.3.5: Let $t_i^{A,1}$ and $t_i^{A,m}$ denote the first step and the last step of transaction ADS segment $\Psi(i, \mathcal{A})$ respectively. Let Q be a consistency preserving partition of D. A schedule z for transaction system T is said to be setwise serial if and only if under z,

$$\forall(\mathcal{A} \in Q)\forall(T_i \in T)\forall(t^A \in z \wedge t^A \in T_i)((t_i^{A,1} > t^A) \vee (t^A > t_i^{A,m}))$$

where t^A represents any step accessing ADS $\mathcal{A}$ in the transaction system T.

Having defined the concept of setwise serial schedules, we now define a setwise serializable schedule as one which is equivalent to a setwise serial schedule.

Definition 5.3.6: A schedule z for transaction system T is said to be setwise serializable if there exists a setwise serial schedule z^* for T such that $z \equiv z^*$.

given a set of consistent states such as (a_1, a_2, a_3), (b_1, b_2, b_3) and (c_1, c_2, c_3), we must prove that $\{a_1, b_2, c_3\}$ is also consistent. First, we apply the intersection of {1} and {1, 2} to "A, B" and "A, C" respectively. States (a_1, b_2, b_3) and (a_1, c_2, c_3) are two of the four new consistent states. Next, we apply the intersection of {2, 3} and {3} to these two new states. One of the two resulting consistent states is $\{a_1, b_2, c_3\}$. We now give a general proof of Lemma 5.3.2-3.

Lemma 5.3.2-3 if P_1 and P_2 are CP, then their least common refinement is also CP.

Proof: Let $P_1 = \{S_1, ..., S_m\}$ and $P_2 = \{\sigma_1, ..., \sigma_n\}$. Their least common refinement is $P_1 \cap P_2 = \{C_1, ..., C_L\}$, where $C_i = S_j \cap \sigma_k$ for some j, k, i = 1 to L.

Let $X_i \in U$, i = 1 to L and $Y \in \Omega$ be given such that $\pi_{C_i}(X_i) = \pi_{C_i}(Y)$, i = 1 to L. We must prove $Y \in U$ to conclude that the $P_1 \cap P_2$ is CP.

To this end, we define a sequence $\{X^*_i, i = 1 \text{ to } L\}$ as follows: $X_1^* = X_1$, $X_j^* = H_{C_j}(X^*_{j-1}, X_j)$, j = 2 to L. Noting that $C_j = S_i \cap \sigma_k$, Lemma 5.3.2-2 indicates that $X_j^* \in U$, j = 1 to L. It follows that $X^*_L = Y \in U$. □

Theorem 5.3.2: There exists a unique maximal CP partition.

Proof: Suppose that there exists more than one maximal CP partition. The least common refinement of distinct maximal CP partitions is CP by Lemma 5.3.2-3, thus contradicting the maximality assumption. □

Corollary 5.3.2: There exists a unique maximum CP partition Q of D.

In the next section, we show how consistency preserving partitions can be used to schedule transactions in a non-serializable fashion. Although any CP partition can be used, Theorem 5.3.2 indicates that there is a CP partition that is "most refined" with respect to a given set of consistency constraints. This partition will allow the maximal concurrency in concurrency control.

Lemma 5.3.2-1: Suppose that $X_1, X_2 \in U$. If S is an element of any CP partition of I, then $H_S: U \times U \rightarrow U$.

Proof: If $S = I$, then $H_S(X_1, X_2) = X_2$, and the result follows.

Let $P = \{S, \sigma_1, ..., \sigma_k\}$, $k \geq 1$ be a CP partition.

Define $W_0 = X_2$, $W_i = X_1$, $i = 1$ to k; so that $W_i \in U$, $i = 0$ to k.

$\pi_S(W_0) = \pi_S(H_S(X_1, X_2))$

$\pi_{\sigma_i}(W_i) = \pi_{\sigma_i}(H_S(X_1, X_2))$, $i = 1$ to k.

Given that P is CP, $H_S(X_1, X_2)$ is therefore in U by the definition of a CP partition. Thus H_S maps pairs of consistent state into a consistent state. □

When we have two or more distinct CP partitions of the same index set I, sets from distinct partitions could intersect. Lemma 5.3.2-2 generalizes Lemma 5.3.2-1 by allowing the intersections to be used for the specification of swapping. For example, let $P_2 = \{ \{1\}, \{2, 3\} \}$ be a second CP partition. The intersection of $\{1, 2\} \in P_1$ and $\{2, 3\} \in P_2$ is $\{2\}$. Lemma 5.3.2-2 states that the two states resulting from swapping the projections of A and B specified by $\{2\}$ are also consistent. That is, (a_1, b_2, a_3) and (b_1, a_2, b_3) are consistent. We show the consistency of (a_1, b_2, a_3) as follows. First, we use Lemma 5.3.2-1 to swap the projections of A and B specified by $\{1\}$ of P_2. One of the two resulting consistent states is $E = (a_1, b_2, b_3)$. Next, swapping the projections of A and E specified by $\{3\}$ of P_2, we find (a_1, b_2, a_3) to be a consistent state also. We now give a general proof of Lemma 5.3.2-2.

Lemma 5.3.2-2: Suppose that $S \in P_1$ and $\sigma \in P_2$, where P_1 and P_2 are CP. Then $H_{S \cap \sigma}: U \times U \rightarrow U$.

Proof: If $S \in P_1$, then there exists a CP partition P such that $S^c \in P$. Since $X_1, X_2 \in U$, it follows that $H_{S^c}(X_1, X_2) \in U$ by Lemma 5.3.2-1. Therefore, $H_\sigma(X_1, H_{S^c}(X_2, X_1)) \in U$ as well. The Lemma follows, since $H_\sigma(X_1, H_{S^c}(X_2, X_1)) = H_{S \cap \sigma}(X_1, X_2)$. □

Lemma 5.3.2-3 demonstrates that the least common refinement of any two CP partitions is also a CP partition. For example, the least common refinement of P_1 and P_2, $\{\{1\}, \{2\}, \{3\}\}$, is also CP. In this case,

$\bigcup_{X \in U}\{\pi_{S_i}(X)\}$. The set of consistency constraints C_i whose truth set is the consistent states of $\mathcal{A}_i$ is called the ADS consistency constraints of $\mathcal{A}_i$. That is,

$$U_i = \{ \pi_{S_i}(Y) \mid C_i(\pi_{S_i}(Y)) = 1 \}$$

Theorem 5.3.1: The conjunction of all the ADS constraints C_i, i = 1 to k, is equivalent to consistency constraints C of D. That is,

$$C = C_1 \wedge C_2 \wedge \ldots C_k.$$

Proof: Let U^* be the truth set of the conjunction of all the ADS constraints. We have,

$$U^* = \{Y \mid C_i(\pi_{S_i}(Y)), i = 1 \text{ to } k\}$$

$$= \{Y \mid \pi_{S_i}(Y) \in U_i, i = 1 \text{ to } k\} = U.$$

Hence, $C = C_1 \wedge C_2 \wedge \ldots C_k$. □

We now prove the theorem that there always exists a unique maximal CP partition. First, we note that CP partitions exist, since the trivial partition { I } is CP. Furthermore, the CP partitions are partially ordered by refinement. That is, for any pair of CP partitions P_1 and P_2, partition P_1 is refined by P_2 if and only if $\forall(S_j \in P_2)\, \exists(S_i \in P_1)(S_j \subseteq S_i)$. A maximal CP partition is one which is refined by no other CP partition. In the following, we will prove that there exists a unique maximal CP partition P. Since the proof is relatively complex, we would like first to illustrate the ideas of the proof.

The proof is based on three lemmas. The idea of Lemma 5.3.2-1 is illustrated by the following example. Let $P_1 = \{ \{1, 2\}, \{3\} \}$ be a CP partition of the index set $I = \{1, 2, 3\}$. Suppose that $A = (a_1, a_2, a_3)$ and $B = (b_1, b_2, b_3)$ are two consistent states. Let S be a partition set, either {1, 2} or {3}. Lemma 5.3.2-1 states that the two new states which result from swapping the projections of A and B specified by S are also consistent. That is, (a_1, a_2, b_3) and (b_1, b_2, a_3) are consistent.

We now define a mapping H_S: $\Omega \times \Omega \rightarrow \Omega$ as follows, where $S \subseteq I$. Given that $X_1, X_2 \in \Omega$, $H_S(X_1, X_2) = Y$, where Y satisfies $\pi_S(Y) = \pi_S(X_2)$ and $\pi_{S^c}(Y) = \pi_{S^c}(X_1)$, where $S^c = I - S$. Thus $H_S(X_1, X_2)$ replaces the projections of X_1 specified by S with the projections of X_2 specified by S.

It is important to note that inter-related data objects can be in different atomic data sets, as long as the consistency of an atomic data set can be satisfied independently of other atomic data sets. For example, we can move a job waiting on the job queue of one computer to the job queue of another computer in order to balance the workloads. That is, these job queues are related by some load balancing transactions. However, at each computer a load balancing transaction must observe the consistency constraints of the local job queue such as the maximum size.

5.3.1 Atomic Data Sets

In this section, we first define the concepts of atomic data sets and a consistency preserving partition of the database. Next, we show that the conjunction of the consistency constraints of the atomic data sets is equivalent to that of the database. Finally, we prove that there is always a unique maximal consistency preserving partition with respect to a given set of consistency constraints.

Definition 5.3.1-1: Let $I = \{1, 2, ..., n\}$ be the index set of database D. The index $i \in I$ specifies the data object $O_i \in D$. Let $\pi_s(Y)$ denote the projection of an n-tuple $Y \in \Omega$ using the set of indices $S \subseteq I$. That is, $\pi_s(Y)$ denotes the tuple whose elements are the values of the data objects indexed by S. Let $P = \{S_1, ..., S_k\}$ denote a partition of I. Let V_i be the set whose elements are the projections of all the consistent states, $X \in U$, onto an arbitrary index set S_i, that is, $V_i = \bigcup_{X \in U} \{\pi_{S_i}(X)\}$. A partition of the index set I, $P = \{S_1, S_2, ..., S_k\}$, is said to be consistency preserving (CP) if and only if,

$$\forall(Y \in \Omega)\{ [\pi_{S_i}(Y) \in V_i, i = 1 \text{ to } k] \rightarrow [Y \in U] \}.$$

Definition 5.3.1-2: An atomic data set $\mathcal{A}_i$ for a CP partition P is the set of data objects specified by $S_i \in P$. The associated partition of data objects in D, Q, is called a consistency preserving partition of D.

The definition of a CP partition states that a CP partition has the property that any choice of the consistent states of the atomic data sets leads to a consistent state of the database. We now introduce the concept of consistency constraints of an atomic data set. Next, we show that the consistency constraints of the database can be decomposed into sets of ADS consistency constraints.

Definition 5.3.2: Let P be a CP partition of I and let $S_i \subseteq I$ be the set of indices which specify the data objects in the atomic data set $\mathcal{A}_i \subseteq D$. Let the set of all the consistent states of an ADS $\mathcal{A}_i$ be U_i =

serializable scheduling rule is the optimal rule when the transaction semantic information is not available for scheduling. Second, the elementary transactions can be in the form of nested transactions. Finally, to enforce generalized setwise serializability, we can use the setwise two phase lock protocol: each elementary transaction does not release any lock on an atomic data set until it has acquired all the locks on this atomic data set; once an elementary transaction releases a lock on an atomic data set, it does not acquire any new lock on the *same* atomic data set. If an atomic data set has the structure of a tree, we can use the tree lock protocol [Silberschatz 80] instead of the setwise two phase lock protocol.

6. Subtask B1-2: Atomic Transaction Theory: Modular Failure Recovery

In this chapter, we summarize our additional work on Subtask B1, the failure recovery part of our atomic transaction theory. The failure recovery theory addresses the problem of guaranteeing the consistency and correctness of concurrency control in the face of system failures. This work is primarily carried out by Dr. Lui Sha and Prof. John P. Lehoczky.

6.1 Introduction

In the previous chapter, Modular Concurrency Control, we have studied the subject of concurrency control without considering the impact of possible system failures. We now extend our work by taking the effect of system failures into consideration. Computer systems can fail in many ways. We limit ourself to the so-called "clean and soft" failures. A failure is said to be clean and soft if the effects of this failure can be modelled by a network of computers some of which stop running and lose the contents of their main memories. However, the database residing in the stable storage remains intact.

The objective of using failure recovery rules is to ensure that concurrency control can be carried out consistently and correctly despite failures in the system. A failure recovery rule specifies the conditions under which the executed steps of a transaction commit or abort. Committing a sequence of executed steps means non-divisibly transferring the result of the computations from the main memory to the database residing in the stable storage. The commit operation represents irrevocable changes made to the database. For example, the recovery rule associated with serializable schedules is known as *failure atomicity*. This rule requires that a transaction either commits or aborts all executed steps at the end of execution. In other words, serializable schedules creates a virtual "executing alone" environment in which the concurrently executed transactions maintain database consistency and produce correct results. At the end of its execution, a transaction will either non-divisibly transfer the results of computation to the database or abort and re-start later. In this way, the consistency and correctness of concurrency control is ensured despite clean and soft system failures. Failure atomicity is widely accepted as the criterion for failure recovery, and many new protocols are being designed to efficiently implement it [Bernstein 83, Mohan 83, Attar 84, Svobodova 84].

Unfortunately, the non-divisibility in the transferring of the computational results of a transaction to the

database places a fundamental limitation on the degree of concurrency in the execution of a transaction system. In order to ensure failure atomicity and avoid cascaded aborts, the results of the computation performed by a transaction must be withheld by some concurrency control mechanism such as locks until the transaction has successfully committed. This is because a transaction can be aborted by failures before it has committed. If other transactions use the results of the partial computation of a transaction, their computations could be invalidated by the abortion of that transaction. In this chapter, we develop a new failure recovery rule, called the *failure safe* rule, which divides a transaction into a partially order set of *atomic commit segments* and commits them segment by segment. This rule is designed for transaction systems using our concurrency control theory described in [Sha 85b, Sha 85a].

Before the formal investigation, we first give an informal overview of our failure recovery approach. According to our modular concurrency control theory, the database is partitioned into atomic data sets and transactions are written in the form of compound transactions, each of which consists of a partially ordered set of elementary transactions. Given a compound transaction, the failure safe rule examines the transaction ADS segments in each of the elementary transactions. A transaction ADS segment consists of all the steps in an elementary transaction that access the same atomic data set. If the steps of a transaction ADS segment are not interleaved with those of others, then the ADS segment is named an *atomic commit segment*. Otherwise, interleaved ADS segments are taken to be a single atomic commit segment. That is, the failure safe rule partitions a transaction into a partially ordered set of atomic commit segments. A transaction must non-divisibly commit each of its atomic commit segments in an order consistent with the partial order of the atomic commit segments in the transaction.

To illustrate our approach, let us consider the compound transaction example which was also used in [Sha 85b, Sha 85a] to illustrate our concurrency control approach. Suppose that transaction Get-A-and-B needs one unit of some resource at node A and another unit at node B. If it cannot have both, then it would rather have none, since the task cannot be run with only one of the resources. The resource heap at each node is modelled as an atomic data set with consistency constraint "$0 \leq$ HeapSize $\leq$ MaxSize", and the transaction is written in the form of a compound transaction, Get-A-and-B, which is illustrated in Table 6-1.

Compound transaction Get-A-and-B has four atomic commit segments, each of which corresponds to one of the four elementary transactions: Get-A, Get-B, Put-Back-A and Put-Back-B, because each of these elementary transactions consists of a single transaction ADS segment. It should be pointed out that failure

```
CompoundTransaction Get-A-and-B;
AtomicVariable Obtain-A, Obtain-B : Boolean;
BeginSerial
  BeginParallel
    ElementaryTransaction Get-A;
    BeginSerial
      Writelock Heap A;
      Take a unit from A if available
      and indicate the result in  atomic variable Obtain-A;
      Commit and unlock Heap A;
    EndSerial;

    ElementaryTransaction Get-B;
    BeginSerial
      WriteLock Heap B;
      Take a unit from B if available
      and indicate the result in atomic variable Obtain-B;
      Commit and unlock Heap B;
    EndSerial;
  EndParallel;

  BeginParallel;
    ElementaryTransaction Put-Back-A;
    BeginSerial
      If Obtain-A and not Obtain-B
      then BeginSerial
             WriteLock Heap A;
             Put-Back the unit of A;
             Commit and unlock Heap A;
           EndSerial;
    EndSerial;

    ElementaryTransaction Put-Back-B;
    BeginSerial;
      If Obtain-B and not Obtain-A
      then BeginSerial
             WriteLock Heap B;
             Put-Back the unit of B;
             Commit and unlock Heap B;
           EndSerial;
    EndSerial;
  EndParallel;
EndSerial.
```

Table 6-1: Compound Transaction Get-A-and-B

atomicity is a degenerate case of our failure safe rule in that it corresponds to the rule in which the entire set of steps of a transaction is taken as a single atomic commit segment.

From a concurrency control point of view, a concurrent execution of a transaction system is modelled by a

schedule. It is shown in this chapter that owing to the preservation of both the internal ordering of steps in transactions and the integrity of transaction ADS segments, the effect of a system failure under the failure safe rule becomes equivalent to changing an existing generalized setwise serializable schedule to another schedule that is also generalized setwise serializable. Thus, under the failure safe rule system failures are *safe* in the sense that the computations recorded in the database are equivalent to the computations resulting from a failure-free execution of the transaction system. For example, suppose that compound transaction Get-A-and-B fails after its elementary transaction Get-A has committed. Since Get-A leaves the database consistent, other transaction will be executed consistently and correctly despite the failure of Get-A-and-B. Since other transactions leave the database consistent, when Get-A-and-B resumes its execution it will also be executed consistently and correctly. When resuming its execution, Get-A-and-B can get another unit of B or return the obtained unit to A, depending on the availability of B.

In this chapter, the organization of studying failure recovery rules is parallel to that of our studying concurrency control rules in the previous chapter [Sha 85b]. We will use many concepts related to concurrency control such as steps, transactions and schedules. All these concepts are formally defined in the previous chapter, and we will not replicate them in this chapter. In Section 6.2, we develop a model of failure recovery rules and define their properties such as consistency, correctness failure safety and optimality. In Section 6.3 we define the failure safe rule and prove that this rule is consistent, correct and modular. In addition, we prove that this rule is optimal for transaction systems scheduled by generalized setwise serializable scheduling rules. Section 6.4 offers some concluding remarks.

6.2 A Model of Modular Failure Recovery Rules

We now develop our model of modular failure recovery rules. We first define the concept of a modular failure recovery rule. Second, we model the effect of clean and soft failures upon concurrency control. We then define the concept of consistency and correctness for a modular failure recovery rule in the face of clean and soft failures. Next, we introduce the concept of safety for a recovery rule. We conclude this section by addressing the issues of re-scheduling aborted transactions.

Conceptually, a failure recovery rule is a specification of the way to commit the executed steps of a transaction. For example, failure atomicity is a failure recovery rule that specifies that at the end of a transaction all the executed steps must be either committed or aborted as a non-divisible unit. In this chapter,

a failure recovery rule is modelled as a function which takes a transaction and partitions it into equivalence classes called *atomic commit segments*. At the end of executing an atomic commit segment, the computational results of this segment must be either committed or aborted as an indivisible unit. A failure recovery rule is said to be modular if it partitions a transaction independently of other transactions in the system. For example, failure atomicity is a modular failure recovery rule that takes the entire set of steps of a transaction as a single atomic commit segment. We now formalize the concept of a failure recovery rule in Definitions 2.1-1 and 6.2.1-2.

Definition 6.2.1-1: Let $\mathbb{T}_m$ denote the set of all the possible consistent and correct transactions with m steps. Let $\mathbb{T}$ denote the set of all the consistent and correct transactions, that is, $\mathbb{T} = \cup_{m=1}^{\infty} \mathbb{T}_m$. Let P_m denote a partition of an m-step consistent and correct transaction into a partially ordered set of transaction step segments called atomic commit segments. Let $\mathbb{P}_m$ denote the set of all the possible partitions of an m-step consistent and correct transaction. Let $\mathbb{P}$ be the set of all possible partially ordered sets of partitions, that is, $\mathbb{P} = \cup_{m=1}^{\infty} \mathbb{P}_m$. A failure recovery rule for a transaction system with n transactions, $\mathfrak{R}_n$, is a function which takes the transaction system of size n and partitions each of the transactions into a partially ordered set of atomic commit segments.

$$\mathfrak{R}_n: \prod_{i=1}^{n} \mathbb{T} \rightarrow \prod_{i=1}^{n} \mathbb{P}$$

Definition 6.2.1-2: A failure recovery rule $\mathfrak{R}$ is a function which takes a transaction system of any size and partitions each of the transactions into a partially ordered set of commit segments.

$$\mathfrak{R}: \cup_{n=1}^{\infty}(\prod_{i=1}^{n} \mathbb{T}) \rightarrow \cup_{n=1}^{\infty}(\prod_{i=1}^{n} \mathbb{P})$$

such that the restriction of $\mathfrak{R}$ to $\prod_{i=1}^{n} \mathbb{T}$ is $\mathfrak{R}_n$, i.e.

$$\mathfrak{R}|\prod_{i=1}^{n} \mathbb{T} = \mathfrak{R}_n, \; n = 1 \text{ to } \infty.$$

Definition 6.2.2: A failure recovery rule $\mathfrak{R}$ is said to be *modular* if and only if $\mathfrak{R}$ independently partitions each transaction in a transaction system into atomic commit segments. That is,

$$\mathfrak{R}_n(T_1, ..., T_n) = (\mathfrak{R}_1(T_1), ..., \mathfrak{R}_1(T_n)), \; n = 1 \text{ to } \infty,$$

where $\mathfrak{R}_n$ is the restriction of $\mathfrak{R}$ to $\prod_{i=1}^{n} T$.

Having formalized the concept of modular recovery rules, we now state our assumption regarding the atomic commit segments produced by those recovery rules. When failure atomicity is adopted, the commit operation ensures that the computations produced by a transaction are either discarded or transferred to the database as a non-divisible unit. In this case, the values stored in the local variables are irrelevant to the commit operation, because the computation has been completed. However, when we commit a transaction segment by segment, we must not only guarantee that the computations produced by an atomic commit segment are non-divisibly transferred to the database, but also guarantee that the values stored in the local variables associated with the commit segment are non-divisibly transferred to the stable storage as well. This is because when an aborted transaction resumes its execution, an executing step may need the values of the local variables associated with those commit segments already committed. We now formalize our assumption regarding the commit and abort operations of atomic commit segments.

Assumption 6.2.1-1: At the end of executing an atomic commit segment, the computations produced by this segment will be either *committed* or *aborted.*

Assumption 6.2.1-2: An atomic commit segment σ is said to be *committed* if and only if

1. the computations produced by this segment are non-divisibly transferred to the database. That is, let the data objects accessed by atomic commit segment σ be O_1, ..., O_k and the values output by σ be $O_1[v_f]$, ..., $O_k[v_f]$. The values of the data objects in the database will be $O_1[v_f]$, ..., $O_k[v_f]$ if σ is committed.

2. The values stored in the local variables associated with segment σ will also be non-divisibly transferred to the stable storage. That is, let L_1, ..., L_k be the set of local variables associated with the transaction steps in σ. Let $\mathcal{L}_1$, ..., $\mathcal{L}_k$ be a set of data objects in the stable storage but not part of the database. Data objects $\mathcal{L}_1$, ..., $\mathcal{L}_k$ are said to be the *private stable storage* for σ. The private stable storage of an atomic commit segment σ can only be written by the commit operation of σ and can only be read by steps of other commit segments in the same transaction. When σ has committed, we have $\mathcal{L}_1 = L_1$, ..., $\mathcal{L}_k = L_k$.

Assumption 6.2.1-3: An atomic commit segment is said to be aborted if and only if its computations are discarded. That is,

1. Let $O_1[v_b]$, ..., $O_k[v_b]$ be the values of data objects O_1, ..., O_k input to atomic commit segment σ. The values of the data objects O_1, ..., O_k are given by $O_1[v_b]$, ..., $O_k[v_b]$ respectively.

2. The private stable storage associated with σ has the initial value "nil" for each of the data objects $\mathcal{L}_1, \ldots, \mathcal{L}_k$. The value "nil" is a special value reserved for recovery management such as the bit pattern of a word with all 1's. "nil" must not be in the domain of any data object. When the private stable storage $\mathcal{L}_1, \ldots, \mathcal{L}_k$ are assigned to a commit segment, their initial values are initialized to "nil". This indicates that they have not been used to store values by the commit segment.

Assumption 6.2.1-4: When resuming the execution of an aborted atomic commit segment $\sigma \in T_i$, all the local variables associated with *committed* atomic commit segments in T_i will be restored to values saved in their private stable storages. That is, let $\sigma_1, \ldots, \sigma_k$ be the atomic commit segments that have been committed. We have

$$\forall(L_i \in \sigma_j, 1 \leq j \leq k)\,(L_i = \mathcal{L}_i)$$

Having defined modular recovery rules and the commit operations for atomic commit segments, we now model the effect of a clean and soft system failure upon concurrency control. When a transaction system is executed according to a schedule z and a failure occurs, schedule z is partitioned into two parts. The part z^e represents the steps in z that have been executed, another part z^f represents those steps yet to be executed.

Definition 6.2.3: Given a schedule z for transaction system T, a clean and soft system failure partitions z into two parts: z^e and z^f. The partial schedule z^e represents steps of z that have been executed prior to the failure, and z^f represents steps in z yet to be executed.

Within an executed partial schedule z^e, there can be some atomic commit segments which have been committed. The computations represented by successfully committed atomic commit segments are modelled by a *committed partial schedule* z^c.

Before we formally define this concept, we must first introduce the useful concept of a *sub-segment*, denoted as "$\sqsubseteq$". We say that a sequence of steps σ_2 is a sub-segment of another sequence of steps σ_1 if and only if all the steps in σ_2 are also in σ_1. In addition, the ordering of steps in σ_2 is consistent with that in σ_1.

Definition 6.2.4: Let σ_1 be a sequence of transaction steps. A sequence of steps σ_2 is said to be a sub-segment of σ_1, denoted as "$\sigma_2 \sqsubseteq \sigma_1$", if and only if

$$\forall((t_i,t_j \in \sigma_2, i \neq j) \wedge (t_i > t_j))\,((t_i,t_j \in \sigma_1) \wedge (t_i > t_j))$$

We now define the concept of a committed partial schedule.

Definition 6.2.5: Let z^c be the executed partial schedule in z. Let σ be an atomic commit segment. The *committed partial schedule* z^c is a sub-segment of z^c that contains only successfully committed atomic commit segments. That is,

1. The committed partial schedule is a sub-segment of the executed partial schedule;

$$z^c \sqsubseteq z^e$$

2. The committed partial schedule does not contain fragmented atomic commit segments;

$$\forall(t \in z^c)(\exists(\sigma \sqsubseteq z^c)(t \in \sigma))$$

3. Transaction commit segments belonging to a transaction are committed in the order defined by the commit rule. We let "$\sigma_1 < \sigma_2$" denote that commit segment σ_1 precedes σ_2.

$$\forall(T_i \in T)\forall((\sigma_1, \sigma_2 \sqsubseteq T_i) \wedge (\sigma_1 < \sigma_2))((\sigma_2 \sqsubseteq z^c) \rightarrow (\sigma_1 \sqsubseteq z^c))$$

Having addressed the issues related to the commit operation, we now turn to the subject of resuming the execution of an aborted transaction. When a transaction fails, some of its atomic commit segments may have been already committed. When the database system resumes the execution of this aborted transaction, we are faced with the problem of scheduling a partial transaction. We assume that a partial transaction is scheduled by the same scheduling rule R as follows. When a partial transaction consists of an integer number of atomic step segments, all we have to do is to ensure that these segments are interleaved serializably with those of other transactions. The main problem in scheduling a partial transaction is that an atomic step segment specified by R might be partitioned by a recovery rule $\mathfrak{R}$ into more than one atomic commit segments. When a failure occurs, it is possible that only some of these commit segments have been committed. In this case, the portion of an atomic step segment left in the remaining schedule z_1 will be taken as an atomic step segment and interleaved serializably with atomic step segments of other transactions in z_1.

For example, suppose that transactionT_i has two atomic step segments σ_1 and σ_2 specified by scheduling rule R. If the entire T_i is in a remaining partial schedule z_1, then these two atomic step segments will be interleaved serializably with atomic step segments of other transactions in z_1. If σ_1 is committed but σ_2 is left in z_1, then σ_2 will be interleaved serializably with those of others in z_1. Finally, suppose that a recovery rule partitions the atomic step segment σ_1 into two atomic commit segments: $\sigma_{1,1}$ and $\sigma_{1,2}$. If only $\sigma_{1,1}$ has been

committed and $\sigma_{1,2}$ and σ_2 are left in z_1, then both $\sigma_{1,2}$ and σ_2 will be taken as atomic step segments in z_1. That is, $\sigma_{1,2}$ and σ_2 will be interleaved serializably with those of others in z_1. We formalize our discussion about scheduling aborted transactions as Assumption 6.2.2.

Assumption 6.2.2: Let z be a schedule for transaction system T satisfying scheduling rule R. Let z_1 be the remaining partial schedule after a failure occurs during the execution of z. Let $\Xi_R(T_i)$ denote the atomic step segments of transaction T_i specified by R.

1. The steps in z_1 are those in z but not in the committed partial schedule z^c.

   $\forall(t)((t \in z) \wedge (t \notin z^c)) \rightarrow (t \in z_1))$

2. If all the steps of a transaction T_i are in z_1, then the atomic step segments of T_i are still represented by $\Xi_R(T_i)$.

3. Let T_i^p denote the partial transaction of T_i in the remaining partial schedule z_1. A partial transaction T_i^p is a sequence of steps such that $(T_i^p \sqsubseteq T_i) \wedge (\forall(t) (((t \in T_i) \wedge (t \in z_1)) \rightarrow (t \in T_i^p))$. Partial transaction T_i^p is scheduled by R as follows. Let $\Xi_R(T_i^p)$ denote the atomic step segments specified by R with respect to the partial transaction T_i^p. We have that

   a. each atomic step segment of the partial transaction is a refinement of some atomic step segment of the original transaction;

      $\forall(T_i^p \sqsubseteq z_1) \forall(\sigma \in \Xi_R(T_i^p)) (\exists(\sigma^* \in \Xi_R(T_i)) (\sigma \sqsubseteq \sigma^*))$,

   b. each step that is in an atomic step segment of the original transaction and in the partial transaction is in some atomic step segment of the partial transaction.

      $\forall(\sigma^* \in \Xi_R(T_i)) \forall((t \in \sigma^*) \wedge (t \in T_i^p)) (\exists(\sigma \in \Xi_R(T_i^p)) (t \in \sigma))$

4. The remaining partial schedule z_1 satisfies scheduling rule R. That is, the atomic step segments specified by R in a (partial) transaction will be interleaved serializably with those of others in z_1.

Having modelled the effect of a single failure, we now address the issue of multiple failures. When the first failure occurs during the execution of z, the committed atomic commit segments are represented by z^c. The executed but not yet committed transaction steps are aborted. These aborted steps are re-scheduled together with steps in z^f to create the *remaining partial schedule* z_1. In the execution of z_1, suppose that a second failure occurs. In this case, we have a new committed partial schedule, z_1^c, and a new remaining schedule z_2. This process continues until all the steps in the transaction system are executed and committed.

Definition 6.2.6-1: Let z be a schedule of transaction system T satisfying scheduling rule R. When the first failure occurs, the committed partial schedule is denoted as z^c, and the remaining partial schedule is denoted as z_1. The k^{th} remaining partial schedule z_k is the remaining schedule for z_{k-1} after the k^{th} failure.

Definition 6.2.6-2: When there are no failures, the execution of a transaction system is modelled by a schedule z. With n failures, the execution of a transaction system is modelled by a committed schedule $z(n) = (z^c, z_1^c, ..., z_n^c)$.

In order to investigate the execution of transaction systems in the face of failures, we must relate the concept of a committed schedule to our established results on scheduling rules.

Theorem 6.2.1: Let a committed schedule for a given total of k failures in the execution of transaction system $T = \{T_1, ..., T_n\}$ be $z(k) = \langle z^c, z_1^c, ..., z_k^c \rangle$, where z_i^c is the i^{th} committed partial schedule after the i^{th} failure.[9] Committed schedule z(k) satisfies scheduling rule R if and only if

1. The ordering of the steps of transaction T_i, $1 \leq i \leq n$, in z(k) is consistent with that in transaction T_i, $1 \leq i \leq n$.

$$[\forall(t)((t \in z(k)) \Leftrightarrow (t \in \cup T))] \wedge [\forall(T_i \in T)\forall((t_{i,j}, t_{i,k} \in T_i) \wedge (t_{i,k} > t_{i,j})) ((t_{i,j}, t_{i,k} \in z(k)) \wedge (t_{i,k} > t_{i,j}))]$$

2. Atomic step segments specified by R and belonging to different transactions are interleaved serializably in z,

$$\exists(z \in Z(T)) ((z(k) \equiv z) \wedge (z \text{ is atomic step segment serial}))$$

Proof: It directly follows from Assumptions 6.2.1-1, 6.2.1-2, 6.2.1-3 and 6.2.1-4, Definitions 6.2.6-1 and 6.2.6-2, and Definitions 5.2.10 and 5.2.12 in the previous chapter. □

Having modelled the effect of failures upon concurrency control, we are now in a position to define the concept of consistency and correctness of a recovery rule. We consider a recovery rule $\mathfrak{R}$ designed for a consistent and correct scheduling rule R to be consistent and correct if and only if $\mathfrak{R}$ ensures the consistency and correctness of the committed schedules in the face of any finite number of system failures.

---

[9] Note that $z_k^c = z_k$ because there are no more failures after the k^{th} failure, and the entire z_k is committed.

Definition 6.2.7: Let $\mathfrak{R}$ be the recovery rule designed for the scheduling rule R, recovery rule $\mathfrak{R}$ is said to be consistent and correct if and only if

$\forall(z \in Z_R(T))\,(z(n)$ is consistent and correct, $0 \leq n < \infty)$

where $Z_R(T)$ is the set of all the schedules for transaction system T satisfying R.

We now define an important property of a recovery rule called *safety*. We consider a recovery rule $\mathfrak{R}$ designed for a scheduling rule R as being *safe* if and only if the committed schedules satisfy scheduling rule R. That is, the computations recorded in the database cannot be distinguished from those resulting from an execution of a schedule $z \in Z_R(T)$ without failures.

Definition 6.2.8: A recovery rule $\mathfrak{R}$ designed for a scheduling rule R is said to be *safe* if and only if for any given schedule $z \in Z_R(T)$ the commit schedules satisfy R, i.e.

$\forall(z \in Z_R(T))(z(n) \in Z_R(T), 0 \leq n < \infty)$

Theorem 6.2.2: If recovery rule $\mathfrak{R}$ designed for a consistent and correct scheduling rule R is safe, then recovery rule $\mathfrak{R}$ is consistent and correct.

Proof: Since all the schedules satisfying scheduling rule R are consistent and correct, it follows from Definitions 6.2.7 and 6.2.8 that $\mathfrak{R}$ is consistent and correct. □

We now turn to the subject of optimality of nodular and safe recovery rules. We measure the concurrency provided by a modular recovery rule by how finely it partitions a transaction. This is because computations produced by an atomic commit segment must be withheld by locks or some other mechanism until this segment has been committed. To allow other transactions to use the results produced by a commit segment before its commit could lead to cascaded aborts – a highly undesirable event. Generally, the finer the partition, the smaller is the size of a commit segment and thus the higher the degree of concurrency.

Definition 6.2.9: Let T be the set of all the consistent and correct transactions. Let $\mathbb{R}$ be the set of all the modular and safe recovery rules associated with some consistent and correct modular scheduling rule R. Let $\Xi_{\mathfrak{R}}(T_i)$ denote the partition resulting from $\mathfrak{R} \in \mathbb{R}$ partitioning $T_i \in T$. A modular and safe recovery rule $\mathfrak{R}$ is

said to be optimal with respect to the associated scheduling rule R, if and only if $\mathfrak{R}$ always produces the most refined partitions. That is, $\mathfrak{R}$ is optimal if and only if

$$\forall(T_i \in T)\forall(\mathfrak{R}^* \in R)\,((\sigma \in \Xi_{\mathfrak{R}}(T_i)) \rightarrow (\sigma \in \Xi_{\mathfrak{R}^*}(T_i)))$$

where σ denotes an atomic commit segment.

We now conclude this section by addressing the issues in re-scheduling aborted transactions. The main point is that an atomic commit segment must be a superset of some atomic step segments.

Theorem 6.2.3: An atomic commit segment produced by a modular and safe recovery rule must be a superset of some atomic step segments produced by the associated scheduling rule R.

Proof: Suppose that this claim is false, and there exists a modular and safe recovery rule $\mathfrak{R}^*$, which divides atomic step segment σ_1 of transaction T_1 into n commit segments with $n \geq 2$. Let the commit segments be $\langle \sigma_{1,1}, ..., \sigma_{1,n} \rangle$. Let the set of data objects read or written by σ_1 be $\mathcal{A}$. Let transaction T_2 consist of only one atomic step segment σ_2 which writes into every data object in $\mathcal{A}$. Now consider a schedule z for transaction system $T = \{T_1, T_2\}$. Suppose that schedule z satisfies R, and we execute T according to z in which T_2 is executed last. Suppose that a failure occurs just after $\sigma_{1,1}$ has been committed and commit segments $\sigma_{1,2}$, ... $\sigma_{1,n}$ are aborted. Let the remaining partial schedule z_1 be $\langle T_2, \sigma_{1,2}, ..., \sigma_{1,n} \rangle$. Suppose that there are no more failures. That is, $z(1)$ is the committed schedule. Note that $z(1)$ does not satisfy R. This is because in $z(1) = \langle \sigma_{1,1}, \sigma_2, \sigma_{1,2}, ..., \sigma_{1,n} \rangle$, σ_1 precedes σ_2 but σ_2 also precedes σ_1 on $\mathcal{A}$. That is, the atomic step segments of T_1 are not interleaved serializably with that of T_2. This contradicts the assumption that $\mathfrak{R}^*$ is safe. □

6.3 The Failure Safe Rule

Having developed a model of modular recovery rules, we now define our failure safe rule and prove that this rule is modular, consistent, correct and safe. We conclude this section by showing that this rule is optimal for transaction systems using generalized setwise serializable schedules.

When we study schedules, a useful concept is the serializable interleaving of the atomic step segments of one transaction with those of other transactions. In the study of failure recovery, an important concept is the interleaving of transaction ADS segments within the same transaction. When transaction ADS segments of a

given transaction are interleaved, they must be committed as a single atomic commit segment in order to preserve the internal ordering of steps in a transaction.

Definition 3.1: Let $\Xi_g(T_i)$ denote the set of transaction ADS segments resulting from a generalized setwise scheduling rule R_g partitioning T_i. A transaction ADS segment $\sigma_i \in \Xi_g(T_i)$ is said to be *interleaved* with transaction ADS segment $\sigma_j \in \Xi_g(T_i)$ if

$$\exists(t \in \sigma_i)(t^1_{\sigma_j} < t < t^m_{\sigma_j})$$

where $t^1_{\sigma_j}$ and $t^m_{\sigma_j}$ are the first and last steps in segment σ_j respectively.

Definition 3.2: Given a transaction $T_i \in \mathbf{T}$ scheduled by a generalized setwise serializable scheduling rule R_g, the *failure safe rule* $\mathfrak{R}_f$ is a function that produces a partially ordered set of atomic commit segments as follows:

1. For each transaction ADS segment σ in an elementary transaction of T_i, name σ as an atomic commit segment if σ is not interleaved with any other transaction ADS segment in this elementary transaction.

2. When two or more transaction ADS segments in an elementary transaction are interleaved with each other, merge them into a single atomic commit segment.

Theorem 3.1: The failure safe rule is modular.

Proof: It directly follows from Definitions 6.2.2 and 6.2.11. □

Theorem 3.2: The failure safe rule $\mathfrak{R}$ is safe. That is, under $\mathfrak{R}$ we have,

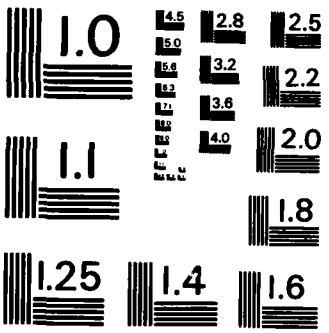
$$\forall(T \subseteq \mathbf{T})\forall(z \in Z_{R_g}(T))\,(z(n) \text{ is generalized setwise serializable, } 0 \leq n < \infty);$$

where $z(n) = \langle z^c, \ldots z^c_n \rangle$ is a committed schedule for a given total of n failures.

Proof: By Theorem 6.2.1, to show that $z(n) = \langle z^c, \ldots, z^c_n \rangle$ is generalized setwise serializable for $0 \leq n \leq \infty$, we need to prove that the ordering of steps in z(n) is consistent with the internal ordering of steps in each of the compound transactions. In addition, all the transaction ADS segments in an elementary transaction must be interleaved serializably with those of others in z(n).

AD-A158 477 RESEARCH IN DISTRIBUTED TACTICAL DECISION MAKING: DECENTRALIZED RESOURCE.. (U) CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE 2/2

UNCLASSIFIED E D JENSEN ET AL. 31 JUL 85 F/G 9/2 NL



To prove that the internal ordering of transaction steps in a compound transaction is preserved , we need to prove that the ordering of the elementary transactions of a compound transaction in z(n) is consistent with that in the compound transaction and that the internal ordering of steps in each of the elementary transactions is preserved. By Definition 3.2, the partial ordering of atomic commit segments is consistent with the partial ordering of elementary transactions in a compound transaction. Since atomic commit segments are committed in an order consistent with the partial ordering of atomic commit segments, the ordering of elementary transactions of a compound transaction in z(n) is consistent with the ordering of them in the compound transaction.

To complete the proof that the internal ordering of steps in a compound transaction is preserved, we need to show that the ordering of steps in each of the elementary transactions is also preserved. Let $t_{i,k}$ and $t_{i,m}$ be two steps in an elementary transaction T_i^e and $t_{i,k} < t_{i,m}$. We need to show that $t_{i,k} < t_{i,m}$ in z(n). There are two cases. First, suppose that $t_{i,k}$ and $t_{i,m}$ are in the same commit segment σ. Since $\sigma \sqsubseteq z(n)$ and the commit segment σ is the superset that contains the transaction ADS segments in which $t_{i,k} < t_{i,m}$, it follows that $t_{i,k} < t_{i,m}$ in z(n). Second, suppose that $t_{i,k}$ is in commit segment σ_x while $t_{i,m}$ is in σ_y and $\sigma_x < \sigma_y$. Since $\sigma_x < \sigma_y$ in z_n, it follows that $t_{i,k} < t_{i,m}$ in z(n).

We now prove that all the transaction ADS segments in an elementary transaction are interleaved serializably with those of others in z(n). First, it follows from Assumption 6.2.2 that steps of elementary transactions are interleaved setwise serializably in z and in the remaining partial schedules z_i, $1 \le i \le n$. Since $z^c \sqsubseteq z$ and $z_i^c \sqsubseteq z_i^e \sqsubseteq z_i$, $1 \le i \le n$, it follows that transaction ADS segments are interleaved serializably with those of others in each of the committed partial schedules z_i^c, $1 \le i \le n$. We now claim that all the transaction ADS segments in an elementary transaction are interleaved serializably with those of others in z(n). If we suppose that this claim is false, then there must exist at least two transaction ADS segments in an elementary transaction such that these two segments are interleaved non-serializably. Let these two ADS segments be σ_1 and σ_2. There are two cases. First, σ_1 and σ_2 are in the same committed partial schedule. This contradicts the result that all the ADS segments of an elementary transaction are interleaved serializably with those of others in the same committed partial schedule. Second, suppose that σ_1 and σ_2 are in two different committed partial schedules z_1^c and z_2^c and that $z_1^c < z_2^c$. By the assumption that these two transaction ADS segments are non-serializable, there must exist some steps $t_{2,j} > t_{2,k}$ in σ_2 and some steps $t_{1,m} > t_{1,n}$ in σ_1 such that "$t_{1,j} < t_{2,m}$" and "$t_{1,k} > t_{2,n}$". However, this contradicts the fact that all the steps in committed partial schedule z_1^c precede those in z_2^c. Hence, all the transaction ADS segments of an elementary transaction are interleaved serializably with those of others in z(n). □

Corollary 3.2: The failure safe rule is consistent and correct.

Theorem 3.3: The failure safe rule $\mathfrak{R}$ is the optimal failure recovery rule within the set of all the modular and safe failure recovery rules for transaction systems using generalized setwise serializable scheduling rules.

Proof: To prove the optimality of the failure safe rule $\mathfrak{R}$, we need to show that the commit segments produced by $\mathfrak{R}$ for any transaction provide the most refined partition. There are two cases. First, a transaction ADS segment of transaction T_i could be taken as a commit segment by $\mathfrak{R}$. By Theorem 6.2.3, this commit segment cannot be further partitioned. In the second case, transaction ADS segments $\sigma_1, ..., \sigma_m$ in an elementary transaction of T_i are taken as a single commit segment by $\mathfrak{R}$, because they are interleaved. Suppose that they are interleaved and taken as a single atomic commit segment σ. Suppose that there exists a modular and safe recovery rule $\mathfrak{R}^*$ which divides σ into more than one atomic commit segments, $\sigma_{c,1}, ..., \sigma_{c,n}$, where $n \geq 2$. By Theorem 6.2.3, each of these commit segments must contain an integer number of transaction ADS segments. Therefore, $\sigma_{c,1}, ..., \sigma_{c,n}$ must be supersets of transaction ADS segments. Let the first failure occur just after the commit of $\sigma_{c,1}$ and let there be no more failures. Suppose that the ordering of steps of T_i in $z(1)$ is consistent with the internal ordering of steps in T_i. This contradicts the assumption that the steps of the transaction ADS segments in σ are interleaved, because in $z(1)$ all the steps of $\sigma_{c,1}$ in z^c precede all the steps in $\sigma_{c,2}, ..., \sigma_{c,n}$ in z_1. It follows that the steps of T_i in $z(1)$ violate the internal ordering of steps in T_i, and therefore $z(1)$ is not generalized setwise serializable. This contradicts the assumption that $\mathfrak{R}^*$ is safe. Thus, the commit segments produced by $\mathfrak{R}$ for any transaction T_i cannot be refined by other modular and safe recovery rules. □

6.4 Conclusion

In this chapter, we have shown that when the transaction scheduling rule is the generalized setwise serializable scheduling rule, the optimal modular and safe recovery rule is the *failure safe rule*. From an application point of view, there are two important points one should be aware of. First, one should write one's compound transaction carefully to avoid the interleaving of transaction ADS segments, whenever this is possible. Unnecessarily interleaving the steps of transaction ADS segments in an elementary transaction could lead to a serious loss of system concurrency. Second, one must realize that the spirit of the failure safe rule is to provide "check-points", so that a transaction can resume its execution after being interrupted by failures. As a matter of fact, once we have committed one single atomic commit segment of a transaction, we can only abort

commit segments not yet committed. We cannot abort the transaction as a whole. Indeed, once we commit a commit segment we are obliged to complete the execution of our transaction. For example, in the compound transaction Get-A-and-B presented in Table 1, once we have obtained one unit of a resource and committed the operation, we are obliged either to get the other unit or to put back the unit which we have already taken.

We give a few informal suggestions regarding the implementation of the failure safe rule. The standard distributed version of the two phase commit protocol[10] [Bernstein 83] can be adopted to commit the atomic commit segments. The major modification is that we also need to store the values of local variables of a commit segment in the stable storage. Saving these values in the stable storage is important for resuming the execution of a transaction that has been interrupted by a failure. When an atomic commit segment commits, the local variables contained in this segment should also be saved in the stable storage. To optimize the storage utilization, we can save only those local variables that are shared by different commit segments. The identification of those shared variables can be made easy if one is willing to request the explicit declaration of them as *atomic variables* in the program of a compound transaction.

[10] Some call it the three phase commit protocol.

7. Plans

Progress on Tasks A and B indicates a strong need to begin an immediate study of the time management problem associated with best effort decision making and transaction facilities. Current technology used for real-time executives (operating systems) is inadequate to support the efficient realization of reconfiguration algorithms and transaction facilities for distributed real-time command and control systems. The development of a theory of distributed scheduling algorithms and its integration with reconfiguration algorithms and a transaction facility has emerged as a critical problem to the success of any real-time decentralized computer system.

Research over the next year on Tasks A and B will continue largely as described in CMU's proposal. Based on the results of research accomplished over the past year, emphasis will be directed at incorporating real-time constraints into transactions and developing scheduling algorithms so that the real-time constraints are met. The expected results from this work should be directly applicable to real-time distributed tactical decision making. Task A will emphasize time-driven resource management --- managing system computation and communication resources efficiently so that real-time constraints will be met. This effort will concentrate on (1) a value function based approach for multi-processor scheduling, (2) scheduling problems of a decentralized system, (3) investigation of a decentralized team decision model, and (4) architectural support for the scheduling of distributed computations. Task B will examine extension of the model of compound transactions to allow for the specification of timing constraints and development of the new theory of co-operating transactions.

References

[Allchin 82] Allchin, James E. and Martin S. McKendry.
Object Based Synchronization and Recovery.
Technical Report GIT-ICS-82/15, School of Information and Computer Science, Georgia Institute of Technology, 1982.

[Attar 84] Attar, R., Bernstein P. A. and Goodman N.
Site Initialization, Recovery and Backup in a Distributed Database System.
IEEE Transaction on Software Engineering, Nov., 1984.

[Bentley 83] Bentley, J. L., Johnson, D. S., Leighton, T. and McGeoch, C. C.
An Experiemental Study of Bin Packing.
Technical Report, Bell Laboratories, Murray Hill, N.J. 07974, 1983.

[Bernstein 83] Bernstein, P. A., Goodman, N. and Hadziacos V.
Recovery Algorithms for Database Systems.
Technical Report, Aiken Computation Laboratory, Harvard University, March 1983.

[Chu 80] Chu, W. W.; Holloway, L. J.; Lan, M.; Efe, K.
Task Allocation in Distributed Data Processing.
Computer 13(11):57-69, November, 1980.

[Chvatal 83] Chvatal, V.
Linear Programming.
W. H. Freeman and Company, 1983.

[Coffman 83] Coffman Jr., E. G., Garey, M. R. and Johnson, D. S.
Approximation Algorithms for Bin Packing - An Updated Survey.
Technical Report, Bell Laboratories, Murray Hill, N. J., 1983.

[DeGroot 74] DeGroot, M. H.
Reaching a Consensus.
Journal of the American Statistical Association 69(345):118-121, March, 1974.

[Dolev 82] Dolev, D.
The Byzantine Generals Strike Again.
Journal of Algorithms 3(1):14-30, March, 1982.

[Eswaran 76] Eswaran, K. P., J. N. Gray, R. A. Lorie and I. L. Traiger.
The Notion of Consistency and Predicate Lock in a Database System.
CACM, 1976.

[Fischer 82] Fischer, M. J.; Lynch, N. A.; Paterson, M. S.
Impossibility of Distributed Consensus with One Faulty Process.
Technical Report, Massachusetts Institution of Technology, September, 1982.

[Frederickson 80] Frederickson, G. N.
Probabilistic Analysis for Simple One and Two Dimensional Bin Packing Algorithms.
Information Processing Letters, Vol.11, No. 4, 5., Dec. 1980.

[Garcia-Molina 83]
Garcia-Molina, H.
Using Semantic Knowledge For Transaction Processing In A Distributed Database.
ACM Transaction on Database Systems, Vol 8, No. 2, June, 1983.

[Graves 70] Graves, G. W. and Whinston, A. B.
An Algorithm for The Quadratic Assignment Problem.
Management Science, Vol. 17, No. 7, Mar. 1970.

[Graves 81] Graves, S. C.
A Review of Production Scheduling.
Operations Research 29(4):646-675, July-August, 1981.

[Hillier 80] Hillier, F. S. and Lieberman, G. J.
An Introduction to Operations Research, 3rd Edition.
Holden-Day Inc., 1980.

[Jeffrey 84] Jeffrey, R. C.
The Logic of Decision, Second Edition.
University of Chicago Press, Chicago and London, 1984.

[Jensen 83] Jensen, E. D.
The Archons Project: An Overview.
In *Proceedings of the International Symposium on Synchronization, Control, and Communication in Distributed Systems*, pages 31-35. Academic Press, 1983.

[Jensen 84] Jensen, E. D.
ArchOS: A Physically Dispersed Operating System.
IEEE Distributed Processing Technical Committee Newsletter, June, 1984.

[Kaku 83] Kaku, B. K. and Thompson, G. L.
An Exact Algorithm for The General Quadratic Assignment Problem.
Technical Report, Graduate School of Industrial Administration, Carnegie-Mellon University, Mar. 1983.

[Karp 72] Karp, R. M.
Reducibility among Combinatorial Problems.
Complexity of Computer Computations.
Plenum Press, New York, 1972, pages 397-411.

[Kaufmann 75] Kaufmann, A.
Introduction to the Theory of Fuzzy Sets.
Academic Press, New York, 1975.

[Kleinrock 76] Kleinrock, L.
Queueing Systems, Vol II, pp 424 - 425.
John Wiley and Sons, 1976.

[Korth 83] Korth, H. F.
Locking Primitives in a Database System.
JACM, Vol. 30, No. 1, January, 1983.

[Kung 79] Kung, H. T. and C. H. Papadimitriou.
An Optimal Theory of Concurrency Control for Databases.
In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 116-126. ACM, 1979.

[Lamport 82] Lamport, L.; Shostak, R.; Pease, M.
The Byzantine Generals Problem.
ACM Transactions on Programming Languages and Systems 4(3):382-401, July, 1982.

[Lesser 73] Erman, L. D., Fennell, R. D., Lesser, V. R., and Reddy, D. R.
System Organizations for Speech Understanding.
The Third International Joint Conference on Artificial Intelligence, August 1973.

[Liu 73] Liu, C. L.; Layland, J. W.
Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment.
Journal of the Association for Computing Machinery 20(1):46-61, January, 1973.

[Lynch 82] Lynch, N. A.; Fischer, M. J.; Fowler, R. J.
A Simple and Efficient Byzantine Generals Algorithm.
In *Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems*, pages 46-52. ACM-IEEE, July 19-21, 1982.

[Lynch 83] Lynch, N. A.
Multi-level Atomicity - A New Correctness Criterion for Database Concurrency Control.
ACM Transaction on Database Systems, Vol. 8, No. 4, December, 1983.

[Marschak 72] Marschak, J. and Radner, R.
Economic Theory of Teams.
Yale University Press, 1972.

[McDermott 80] McDermott, D.; Doyle, J.
Non-Monotonic Logic I.
Artificial Intelligence 13:41-72, 1980.

[McDermott 82] McDermott, D.
Non-Monotonic Logic II.
Journal of the Association for Computing Machinery 29(1):33-57, January, 1982.

[Mohan 83] Mohan, C.
Efficient Commit Protocol for The Tree Process Model of Distributed Transactions.
IBM Research Report, RJ 3881 (44078), Computer Science, June, 1983.

[Mohan 85] Mohan, C., Fussell, D., Kedem Z. M. and Silberschatz A.
Lock Conversion in Non-Two-Phase Locking Protocols.
IEEE Transaction on Software Engineering, Jan., 1985.

[Papadimitriou 84]
Papadimitriou, C. H. and Kanellakis, P. C.
On Concurrency Control by Multiple Versions.
ACM Transaction on Database Systems, Mar., 1984.

[Sahni 76] Sahni, S. K.
Algorithms for Scheduling Independent Tasks.
Journal of the Association for Computing Machinery 23(1):116-127, January, 1976.

[Schwarz 82] Schwarz, Peter M. and Alfred Z. Spector.
Synchronizing Shared Abstract Data Types.
Technical Report CMU-CS-82-128, Department of Computer Science, Carnegie-Mellon University, 1982.

[Schwarz 84] Schwarz, P.
Transactions on Typed Objects.
PhD thesis, Department of Computer Science, Carnegie-Mellon University, 1984.

[Sha 85a] Sha, L.
Modular Concurrency Control and Failure Recovery -- Consistency, Correctness and Optimality.
PhD thesis, Department of Electrical and Computer Engineering, Carnegie-Mellon University, 1985.

[Sha 85b] Sha, L., Lehoczky, J. P. and Jensen E. D.
Modular Concurrency Control and Failure Recovery --- Consistency, Correctness and Optimality; Part I: Concurrency Control.
Submitted for publication, 1985.

[Silberschatz 80] Silberschatz, A., and Z. Kedem.
Consistency in Hierarchical Database Systems.
JACM 27:1, 1980.

[Stankovic 83] Stankovic, J. A.
Bayesian Decision Theory and Its Application to Decentralized Control of Job Scheduling.
February, 1983.
Internal documentation, University of Massachusetts, Department of Electrical and Computer Engineering.

[Stefik 82] Stefik, M.; Aikins, J.; Balzer, R.; Benoit, J.; Birnbaum, L.; Hayes-Roth, F.; Sacerdoti, E.
The Organization of Expert Systems, A Tutorial.
Artificial Intelligence 18:135-173, 1982.

[Svobodova 84] Svobodova, L.
Resilient Distributed Computing.
IEEE Transaction on Software Engineering, May, 1984.

[Tokuda 85] Tokuda, H., Clark, R. K. and Locke, C. D.
Archons Operating System (ArchOS) --- Client Interface, Part II.
Technical Report, Department of Computer Science, Carnegie-Mellon University, 1985.

[Weihl 84] Weihl, W. E.
Specification and Implementation of Atomic Data Types.
PhD thesis, Massachusetts Institute of Technology, 1984.

[Wulf 81] Wulf, W. A.; Levin, R.; Harbison, S. P.
HYDRA/C.mmp: An Experimental Computer System.
McGraw-Hill, Inc., 1981.

[Zadeh 79] Zadeh, L. A.
A Theory of Approximate Reasoning.
Machine Intelligence 9.
Wiley, New York, 1979.

END

FILMED

10-85

DTIC